REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing this collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number. PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.

1. REPORT DATE *(DD-MM-YYYY)* September 25, 2010	**2. REPORT TYPE:** FINAL REPORT	**3. DATES COVERED** *(From - To)* May 1,2007 – November 30,2009

4. TITLE AND SUBTITLE
SERVER LEVEL ANALYSIS OF NETWORK OPERATION

UTILIZING SYSTEM CALL DATA

5a. CONTRACT NUMBER FA9550-07-1-0453

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
Victor A. Skormin, PhD

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Center for Advanced Information
Technologies, ECE Department,
Watson School,
Binghamton University
4400 Vestal Parkway,
Binghamton, NY 13903

8. PERFORMING ORGANIZATION REPORT NUMBER

Final Report - 0453

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Air Force Office of Scientific Research
875 North Randolph Street
Suite 325, Rm 3112
Arlington, VA 22203

10. SPONSOR/MONITOR'S ACRONYM(S)
AFOSR

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT

Available to general public

13. SUPPLEMENTARY NOTES

14. ABSTRACT

Malware could be detected by the detection of its self-replication activity. A database of self-replication patterns in the system call domain, such as system call traces of propagation engines of computer worms, was established. Procedures extracting these patterns dispersed within a voluminous system call sequence have been developed on the basis of Colored Petri nets. Host-based IDS utilizing this approach for detecting not only known, but also new, previously unknown malicious programs has been implemented and tested. Host-based system call IDS can lead to false alarms and do not allow to see the "big picture" that is important in the case of a distributed attack. Server-level aggregation and analysis of host-level IDS data can address these problems. A technology for monitoring and preprocessing system calls at the host level, reporting the resultant information to the server, and analyzing the collected information at the server level has been developed. This technology, resulting in the early detection/mitigation of information attacks has been successfully implemented and tested at the computer network testbed.

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			19b. TELEPHONE NUMBER *(include area code)*
U	U	U	UU		607-777-4013

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std. Z39.18

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1.0 OVERVIEW

Modern computer networks and the entire Internet are increasingly vulnerable to information attacks targeting the availability of computer resources and compromising data integrity and confidentiality. Contemporary information assurance is provided not through provable security but practical security: making a system resistant to attack by exceeding the time and resource capabilities of an attacker. Thus, the odds are in favor of the skillful and motivated attacker. Malicious software (malware) is used in the majority of these attacks, and while the technology to detect individual malicious programs has begun to mature, very little progress has been made in the detection of attacks in which multiple malicious programs work together to break the security of a system, typically in an extremely stealthy manner.

The objective of this research is the development of a prototype system for the detection of multipartite malware and multistage attacks, which attack from multiple points within an information infrastructure, simultaneously (multipartite) or in sequential steps (multistage). Multipartite malware is emerging in response to the growing ability to detect individual malicious programs. Multistage attacks have been around for some time but are becoming increasingly prevalent due to improved security. It is imperative that these trends be countered now, before they can establish a strong foothold, for both are extremely insidious forms of attack that can be made almost arbitrarily stealthy, and therefore very difficult to detect.

This research leverages the PIs' prior experience in developing a novel method for the detection of self-replicating malware, which simplifies the way in which these attacks are detected and enables the detection of previously unknown attacks without sacrificing detection accuracy. Detection is based on what programs are actually doing, rather than what they might do. Thus, while it does not prevent the attack from inflicting localized damage within the information infrastructure, it prevents epidemics that result in catastrophic damage. This research is a natural extension to this work in which the activities of programs across an entire information infrastructure are monitored in order to detect attacks against multiple points. Its leads not only to an approach for detecting multipartite and multistage attacks, but will advance our understanding of these emerging forms of malware and counter this insidious threat before it has a chance to become the preferred method used by attackers.

The research efforts reported herein were aimed at the further enhancement of computer network security systems, and were naturally geared towards the enhancement of lower-level IDS (Intrusion Detection System) detecting attacks on the individual hosts, and the development of an upper-level IDS operating on the server level and aggregating the information reported by individual hosts. This enabled us to perform the detection of advanced attacks, and at the same time increase the dependability of detection decisions by significant reduction of false alarms.

Consequently, we developed, implemented and evaluated a technology for monitoring, processing, and analyzing system calls at the host level, reporting the resultant information to a server, and correlating and analyzing the collected information at the server level. This technology facilitated early detection/mitigation of multistage and multipartite stealthy (low and slow) information attacks thus enhancing the security of information infrastructures. Monitoring of systems calls at the host level has been a two-pronged approach, taking advantage of known attack techniques (misuse detection) while facilitating the detection of new methods of attack (anomaly detection). Based on the prior success in the detection of self-replication behavior within sequences of system calls at run-time [1], [2], we continued this effort to search for a broad class of known malicious behaviors in programs. Such detection of malicious behaviors provided accurate information facilitating early detection of multipartite and multistage attacks at the server level. Leveraging the same capabilities developed in the processing of systems calls for the detection of malicious behavior, we developed functional behavior profiles of *legitimate* applications and services running on the hosts. These profiles, based on the identification of program functional blocks derived from raw system call sequences utilizing known interrelationships among their parameters. At runtime, processes were verified against the corresponding profile to detect deviations from legitimate behavior, which could indicate that the process have been subverted for malicious purposes. This task became feasible only due to the full exploitation of the relationships between parameters (attributes) of the dispersed individual system calls forming a logical sequence, previously established by the PIs [1].

Detection of malicious activity at individual hosts is not sufficient for the detection of multistage and multipartite malware. Therefore, we performed monitoring the host-level alerts and process behaviors at the network-level on a server dedicated to the task of analyzing and correlating the information. The server, having access to the “big picture” of activities across the entire information infrastructure, technically, is capable of detection and correlation of the events facilitating the reconstruction of the early stages of incipient attacks against multiple points in the network. In addition, the information processed by the server can be archived for later forensic analysis of attacks or vulnerability analysis in the network.

The developed approach is the first of its kind to consider system call monitoring across an entire information infrastructure. While information is accessed and manipulated by individual processes, whether legitimately or maliciously, the system call domain represents the logical choice for monitoring the overall operation of the network, engaged in legitimate or malicious regimes. This research is novel in its primary objective of stopping multipartite malware and multistage stealthy attacks, which are emerging forms of attack whose potential for stealth threatens our information infrastructure. Aimed at the detection of the specific behavior patterns within the network at large, it has a potential for the detection of the developing previously unknown attack scenarios.

1.1. Detection of Information Attacks in the System Call Domain

Unification and scalability of modern computer systems requires a complex computer software/hardware infrastructure. This infrastructure is facilitated by a computer operating system, which abstracts details of the hardware from application software. Applications (programs) interface with the operating system through the Kernel Application Programming Interface via *system calls*. Therefore, system calls uniquely characterize the behavior of both malicious and legitimate computer programs, providing unambiguous information on what the program actually does. While all forms of malware require support of a running process, system call monitoring facilitates an ultimate method of detection.

Applications and service processes can be divided into several distinct operational phases. For example, our preliminary experiments have identified the following distinguishable operation modes of Internet Explorer: application loading, browsing (loading pages from the Internet) and downloading (retrieving large files). A process might not have one unique overall profile of system calls due its varied functionality and unknown input conditions. However, during any one of its execution phases a process implements several (not many) high level functions which are realized by similar sequences of system calls. Being designed for a specific purpose, these high level functions occur with some regularity. Thus, even under varying input conditions the histogram of system calls across a particular window of time would be consistent for a given phase and unique to the application thus providing dependable information on the functionality of the software that is being executed.

To demonstrate the feasibility of the system calls-based intrusion detection, preliminary experiments at monitoring the system calls made by various processes have been performed. These experiments used the legitimate processes Internet explorer, Notepad, and CCAPP (a non-graphical Norton Antivirus Process). As an example of malware, the forth generation of the Sasser worm (W32.Sasser.D.Worm), which exploits a buffer overflow vulnerability in LSASS (Local Security Authority Subsystem Service), a system process that handles Microsoft Windows security mechanisms (local security and login policies). In the experiments, 600000 system calls were monitored for legitimate processes and 50000 for Sasser Worm. Figure 1 depicts randomly chosen histograms of Internet explorer in its downloading phase and of Notepad in its editing phase. One can see that histograms of system calls issued by Explorer during the downloading phase are consistent (as are those of Notepad) while being drastically different from those of Notepad. The figure also shows Receiver Operation Characteristic for classification between Internet Explorer and Notepad. Classification made using the Mahanalobis distance metric. Training (basis) set consisted of 1000 histograms. By changing distance threshold the corresponding false positive and detection rates were derived. The curve shows high detection rate and low false positive for wide range of thresholds, which indicates dependable performance over a wide range of operation.

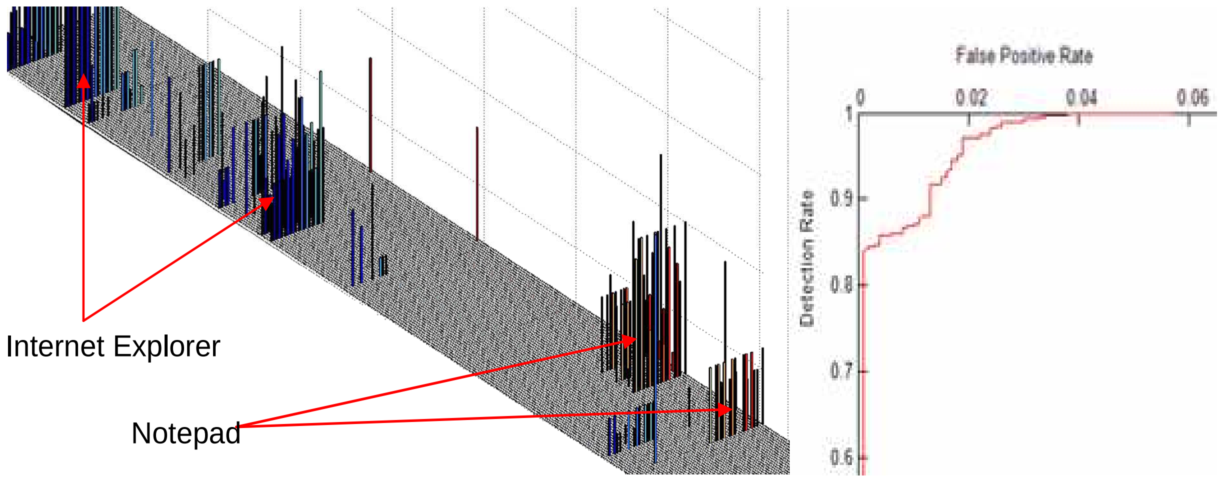


Figure 1. Histogram of System Calls Issued by Internet Explorer and Notepad and Receiver Operation Characteristic for Classification between them Using Mahalanobis Distance

Figure 2 shows a similar set of histograms for Internet Explorer, CCAPP and the Sasser Worm. One can see that worm has even higher consistency with respect to the system calls issued, which can be explained by the repeated use of the same piece of code in many threads in order to boost the speed of propagation. CCAPP does not have graphical interface and carries out certain functions similar to those of Windows services. Hence, it can be expected that most windows services will have highly consistent name distribution.

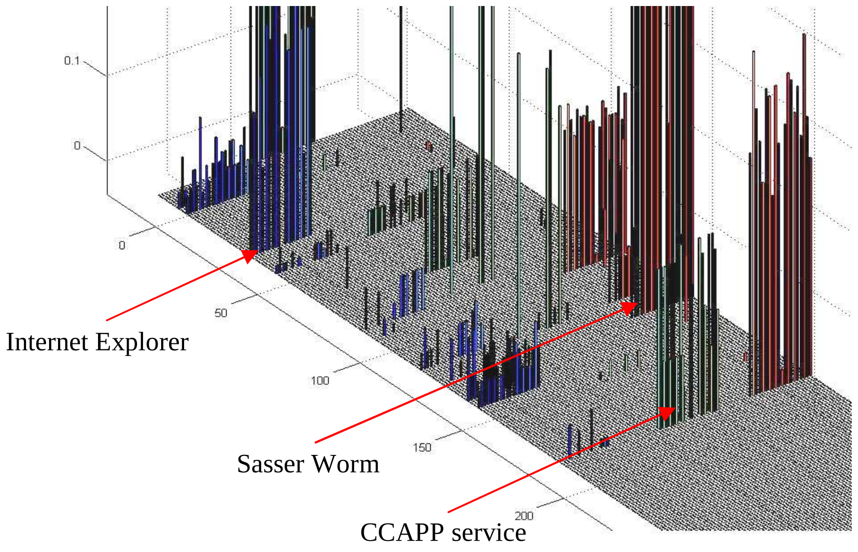


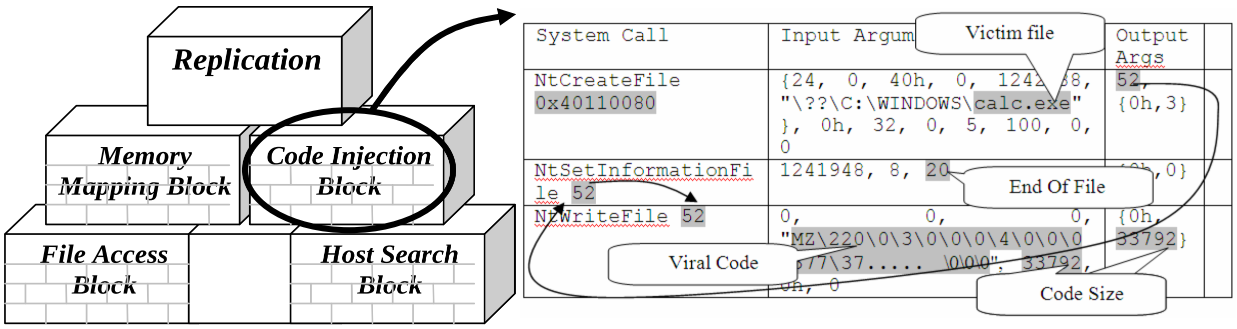
Figure 2. Histograms of System Calls Issued by Internet Explorer, Sasser Worm, and CCAPP.

These preliminary results suggest that system call-domain models of the operating modes for every legitimate application/service can be obtained and used to create a signature of the process. By monitoring the system calls of a process at run time, the functional modes of the process can be verified against these signatures to detect when the process has been taken over by a malicious program.

1.2. Prior Work

An earlier completed research project under the AFOSR funding has demonstrated that malicious software, including executable and encrypted executable codes, could be detected by the detection of its self-replication activity [2]. While the number of malicious computer programs that could be written is infinite, the number of ways to implement self-replication is very limited. Developers of malicious programs are destined to use the same self-replication techniques again and again. Consequently, the developed approach is capable of detecting not only known, but also new, previously unknown malicious programs.

Figure 3 illustrates how self-replication is detected in the system call domain. The self-replication activity of malicious codes manifests itself by generating one of a few very specific sequences of system calls. These system calls can be linked into functional blocks by certain relationships between their input and output parameters. We have developed a database of self-replication patterns defined in terms of these blocks. A procedure that extracts self-replication patterns potentially dispersed within a voluminous system call sequence has been established. This procedure takes full advantage of almost forty attributes accompanying every system call that have mostly been ignored by previous researchers. We have developed a Dynamic Code Analyzer (DCA) that performs simultaneous monitoring of all processes executed by a computer, forms sequences of system calls into functional blocks and composes these blocks, if present, into self-replication patterns. The DCA displays names of the processes engaged in self-replication and suspends those that have almost completed one of the known self-replication routines. Then the user is given the authority to terminate or release the suspended process. Therefore, our DCA may not prevent a malicious code from inflicting damage on a particular computer, but it prevents its spread that causes computer epidemics. The DCA consumes not more than 5% of the computer resources and could be "afforded" by most users in the capacity of a resident security tool complimenting the existing antivirus software. Currently, the DCA is subjected to testing and fine-tuning.



System Call	Input Argum	Output Args	
NtCreateFile 0x40110080	{24, 0, 40h, 0, 1242?38, "\??\C:\WINDOWS\calc.exe" }, 0h, 32, 0, 5, 100, 0, 0	52, {0h,3}	
NtSetInformationFile 52	1241948, 8, 20	{0h,0}	
NtWriteFile 52	0, 0, 0, "MZ\220\0\3\0\0\0\4\0\0\0 ?77\37..... \0\0\0", 33792, 0h, 0	{0h, 33792}	

Figure 3. Example of Functional Blocks Forming a Self-Replication Method

Though this research has been successful at detecting both known and new forms of viruses, the results do not directly carry over to multistage attacks or multipartite malware. Thus, while monitoring system calls within a particular host computer is very informative, it has several limitations. First, it does not allow the comprehensive network-wide “big picture” view necessary to detect and stop multistage attacks. Second, the advanced processing needed to detect multipartite malware can consume resources beyond those available for detection. In addition, the individual entities making up the multipartite malware might exist on different hosts, rendering host-based detection approaches useless against them.

1.3. Research Objectives

This research was aimed at the development of a system in which the monitoring of system calls is performed at the level of individual hosts and the resulting information is reported to a server where it can be coalesced into a suitable form to support network-wide administration and decision making. It is important to emphasize that the information reported to the server does not contain “pure system calls”, but only the alerts generated by the system call-based IDS deployed on particular hosts. This approach facilitates not only the correlation of alerts produced by the system call monitors at individual hosts, but also the correlation of the events observed in the network at large over some arbitrary period of time thus enabling the detection of incipient multistage and multipartite attacks. Since information reported to the server comes with a timestamp, host identification, and event data, forensic analysis of an attack can be performed to aid in vulnerability analysis of the network. In addition, detection accuracy can be improved and false positive rates reduced via correlation of alerts at the server.

The overall structure of the proposed system is shown in Figure 4. Each monitored host computer runs a system call-based IDS including Application Behavior Modeling software, which performs per process system call analysis, host level system call aggregation, and transmission of the information to the server. The dedicated network analysis server receives the information from each monitored host, performs advanced host level analysis and network-wide alert correlation, and performs alerting and archiving functions. Each of these components is discussed in more detail below.

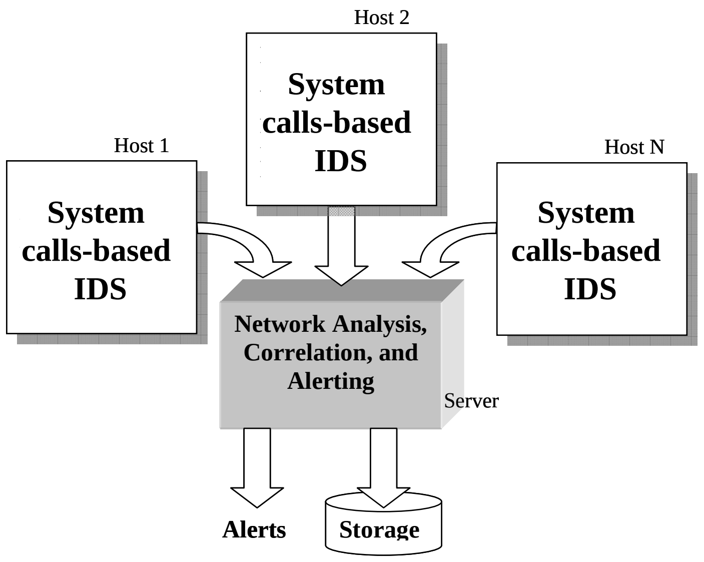


Figure 4. Overview of network-based system call monitoring and analysis approach

The IDS deployed on each monitored network host is based on the earlier developed dynamic code analyzer detects the self-replication activity. The unit performs simultaneous monitoring of all processes executed by the computer, forms sequences of system calls and extracts, if any, known patterns of malicious activities such as self-replication. In addition to expanding the search for malicious activity beyond self-replication, the unit will also dynamically extract the "signature" from each running process to be compared to known signature models. All of this information is aggregated and transmitted to the server for correlation and storage.

Malicious software performs operations that adversely affect stored data and various hardware/software system components. When a legitimate process is hijacked by malware, it begins to execute under its command, issuing system calls that are uncharacteristic of the legitimate process. In terms of system calls issued, the signature of the application deviates from the known model and can be detected. The developed technique represents a two-pronged approach which takes advantage of both normal models of legitimate software (anomaly detection) and known behavior of malware (misuse detection). Traditionally, most research efforts consider these approaches separate, espousing the benefits of one over the other. However, they are complementary approaches and together can significantly enhance the early detection of malicious activity. While searching for known patterns of self-replication activity in executing processes is an extremely effective method for detecting viruses and worms, there are a vast number of operations that can be considered malicious, however, and one cannot always predict in advance every insidious technique that an attacker can use. On the other hand, one can comprehensively model the behavior of a known application and detect deviations from this model, but certain malicious behaviors might be difficult to detect. A comprehensive solution uses the two methods in a complementary fashion to maximize detection thus minimizing the probability of success of the attack.

The sequences of system calls issued during a program's execution define a signature for the program. In modern operating systems, the system call interface represents a barrier between programs and the computer's resources, providing uniform access to whatever hardware lies beneath. In all cases, system calls represent a direct time line sequence of events, which can be analyzed during the execution. For any given process, this sequence can be large or relatively small depending on what system resources it is trying to access. As a result, computer programs are uniquely and completely identified by their sequences of system calls and associated parameters. It is expected that, except for the simplest of programs, the overall sequence of systems calls generated during any given execution is likely to be different. However, these sequences of system calls contain some critical subsequences that define the computational essence of the program. They occur throughout the program execution and could be viewed as a signature uniquely representing the program. This signature can be detected at runtime thus implementing a passive watermarking or verification approach. These signatures can be established for all of the authorized software to be executed on the networked hosts and verified at the host or at the server, depending on the computational intensity of the verification process.

The structure of the resultant IDS is shown in Figure 5. System calls are intercepted by an operating system driver that hooks into the system call stack. This driver outputs for each system call issued a tuple representing the process identifier, system call identifier, parameters of interest, and a Boolean success/failure outcome indicating whether the system call was successful or not. The functional block generator separates the system calls into per-process streams and forms them into higher level functional blocks. Application programmers use application programming interfaces (APIs) and high-level-languages (HLLs) to write software. During compilation, these programs are translated into machine code, at which time the high-level operations are divided into smaller pieces that can be implemented with specific system calls. By themselves, system calls do not provide much information about what a program is doing. When linked through their arguments, however, they can be aggregated back into a form that provides a representation of the higher-level program functionality. Although the system does not necessarily aggregate back into the original source code form, the functionality represented by the higher-level block does provide an indication of the behavior of the program. Therefore, in a sense the two forms are equivalent.

The dynamic code analysis and application behavior modeling use these functional blocks as input to implement the two-pronged detection approach. Dynamic code analysis matches the higher level functional blocks representing program operations against known malicious activities (such as self-replication). This implements the misuse detection function at the host, which looks for known instances of malicious activity. Application behavior modeling performs the anomaly detection that is the complement of the dynamic code analysis. This block compares the dynamic profile of system calls from each process to the known behavior (signature) of the application. Alerts, indicating the presence of a particular malicious activity or deviations from known behaviors of the program, can trigger corrective action at the host; in addition, they are forwarded to the aggregator to be sent to the server for advanced correlation and analysis.

When attempting to elude anomaly-based approaches, attackers sometimes resort to mimicry attacks, where they try to mask the anomalous behavior by hiding it inside of a significant amount of normal-looking activity. In the system call domain, however, this is a non-trivial task. System calls are not isolated, random entities that can be issued arbitrarily. System calls have input and output data which must follow certain rules and convention or the system call will return unsuccessfully. One of the parameters to be monitored in this research is the Boolean "success/fail" outcome of every system call. The signature of a normal application will include a certain mix of successful system calls. By including this parameter in the application signature the work to be done in performing a mimicry attack is made significantly harder because the attacker is prevented from simply issuing a certain mix of system calls- they must also perform the behavior of the "host" application.

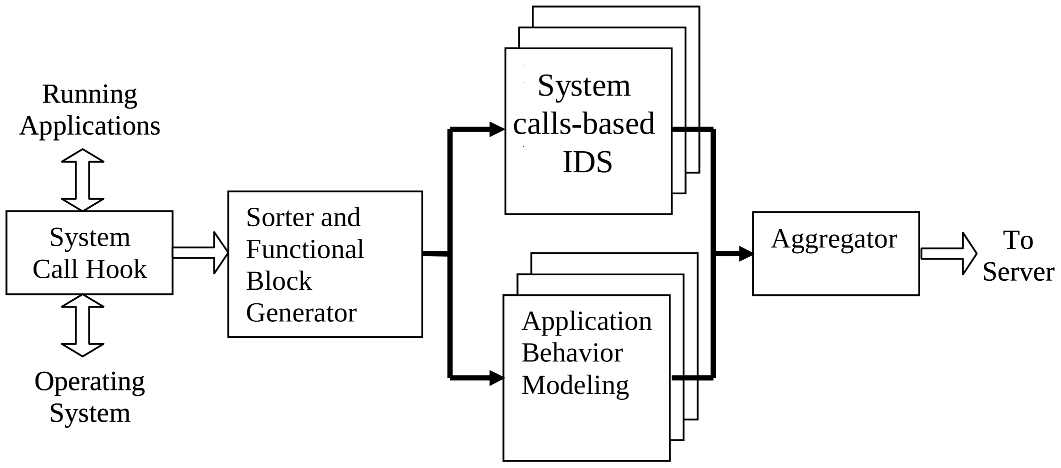


Figure 5. Dynamic Code Analysis and Aggregation

One can realize that there are a vast number of operations that can be considered malicious and could be detected within the sequence of system calls. In addition, the sequence of system calls produced by an application can be huge and the malicious operation can be dispersed throughout the sequence, or intentionally inserted in a stealthy manner making run-time detection a non-trivial task. An attacker who utilizes this form of attack is indeed a skillful one; therefore, this is the form of attack that is most crucial to detect. The proposed two-pronged approach makes it difficult for an attacker to utilize this attack mode and go unnoticed. The normal activity will be classified and filtered out by through verification of the program signature, leaving the suspicious activity. If the attack utilizes known behaviors, these can be detected separately by the code analysis. Thus, the attacker must get through two forms of detection, making his job significantly more difficult.

Dynamic code analysis and application behavior modeling alone are effective at detecting certain forms of malicious software, but their detection abilities are inherently limited by the host-specific information they have access to. A network-wide view is necessary to detect multistage and multipartite attacks. The high-level analysis and correlation required to detect these types of attacks requires the computational resources of a server dedicated to this task. In addition, having this function contained in a dedicated server allows the server to be hardened against attackers who intend to circumvent detection by attacking the detection mechanism itself.

It is proposed to develop a network analysis and correlation server dedicated to the task of detecting multipartite and multistage attacks. This server will receive the system call blocks that have been formed at each monitored host as well as alert data. This information appears as multiple ordered streams of events which are correlated and analyzed. Correlation of alerts implies inferring interrelationships among alerts produced at individual hosts. This can be accomplished using a priori knowledge about known attack strategies or through real-time clustering of alerts. The former approach is suitable for detection of known multistage attacks only. Although the PIs do not rule out using a priori knowledge of attack strategies to correlate alerts, they believe that the need to detect new attacks through application profiling (anomaly detection) necessitates the use of a probabilistic clustering methodology.

It should be understood that reporting system call information from hosts to the server can overload the communication channels of the network, and therefore, the volume of the reported information must be reduced to the necessary minimum. In the proposed system, the information transmitted between hosts and server is not raw system call data but higher-level program operations that have been created by joining together these system calls. Thus, the amount of traffic depends on the throughput of these larger functional blocks. In addition, previous experience with system calls analysis has shown that the overwhelming majority of system calls issued in the Windows OS are those used for graphics. In the proposed approach, consecutive runs of these system calls are joined into functional blocks, which greatly increases the amount of compression obtained. Thus, the proposed approach will not overload the communications network.

The information collected by the server will be stored in a database for later retrieval to facilitate forensic analysis and vulnerability analysis. This database will contain all alerts generated at hosts as well as the server. In addition, the individual system call streams (aggregated into functional blocks) will be stored using some form of lossy compression that facilitates it subsequent analysis with requiring enormous storage space. Bloom filters have been applied for this purpose. The chose method, however, will be highly dependent on the form of the data and will be studied.

2.0 HOST-BASED DETECTION: UTILIZING PROPAGATION BEHAVIOR OF COMPUTER WORMS

2.1. Introduction

Our ever-growing dependence on Internet is accompanied by ever-growing concerns about the networks vulnerability to information attacks and the dependability of the existing network security systems. Major threats, well recognized by government, private institutions and individual users, are stemming primarily from self-proliferating malicious software such as network worms.

Network worms perpetrating remote code execution attack, such as buffer overflow, stack overflow, heap overflow, etc., have two vital components, propagation engine (shell code) and exploit. The shell code being a necessary part of the propagation engine is executed by the vulnerable process just after the exploit vector rerouted the control flow. The shell code creates specific conditions which are utilized by the attacking worm to complete the propagation session. Hence, every network worm performing remote code execution attack, employs a particular type of propagation engine and corresponding shell code in every attack to replicate itself into the victim machine.

The adversaries usually utilize standard propagation engines along with available exploits of the selected vulnerability. This could be explained by the fact that reverse-engineering of the services, determining vulnerability, developing the exploit and producing specific shell code (propagation engine) requires special experience and knowledge which is possessed by a small community of computer professional. The largest part of the worms is written by so-called "script–kiddies" [1], who utilize available exploits with standard shell codes as buffers in the attack packets and implement their own payload what results in the new worms.

Moreover, a worm family may spawn many strains, so-called versions of the original worms. These strains are created in order to avoid matching to existing binary signatures of known anti-virus databases and to prolong the life-cycle of the worm. While recompiling the original worm code into novel strains, the adversaries change image name, synchronization object name, registry records and employ equivalent code modification techniques. However, according to our experience, versions of one worm family tend to share one or two propagation engines. For instance, W32.Sasser worm has seven known strains, while W32.Padobot worm has 29 strains and in spite of their mutations they utilize only two propagation engines. This is a good demonstration of the fact that the source code of the worms could be easily subjected to change to break known simple signatures, but adversaries hardly change shell coder buffer since it requires to program in specific, base-independent style[1] what involves much more efforts. Even

[1] It requires computing delta offset, searching for entry address of API functions in DLLs and performing variable address alignment.

if an effort is made to change the shell code (propagation engine) to make it binary different, one has to preserve the initial system call pattern to achieve the propagation effect.

As a result, most of the network worms share the same trivial propagation engines and shell codes. To achieve the propagation effect, the shell code must invoke system calls through API functions to utilize operation system (OS) resources. Consequently, shell codes of the worms having the same propagation engine have the same realization in the system call domain. Therefore, the task of detecting worms can be narrowed down to recognizing the standard propagation engines utilizing the system call signatures.

Based on the highly successful dynamic code analyzer (DCA) concepts [2], the authors developed and evaluated the Propagation Engine Detector (PED) system capable of detecting attacks perpetrated by network worms. This system detects the worm shell code activity performed by a process during the attack session. Moreover, PED recognizes the type of propagation engine employed by the worm exploiting the process. The developed PED system utilizes Colored Petri nets (CP-net) to trace in parallel interrelated chains of system calls issued by the monitored process to recognize high level API functions invoked by the process. The detected high level functions are analyzed, not individually but in combinations, to determine how the process creates and manipulates objects of the operation system. Such information is finally processed by CP-net to detect if the process activity exhibits the functionality of the particular types of propagation engines.

2.2. Background and Related Work

System call-based Intrusion Detection Systems (IDS) utilize two main approaches, misuse detection and anomaly detection. Misuse detection or so-called signature-based detection systems employ known traces of system calls to detect malicious activity. This approach ensures high level of accuracy, but fails to detect previously unknown attacks. Anomaly-based detection utilizes models of normal behavior of legitimate and especially privileged processes with respect to invoked system calls. In the detection mode, these systems check consistency between invoked system calls and the profile of normalcy for a given process. Anomaly-based IDS are able to detect unknown (new) attacks, but suffer from high rate of false positives.

Up to now a number of anomaly-based as well as misuse-based IDS have been proposed. These systems could trace merely the order of system call execution. The efficiency could be further enhanced by analyzing arguments of system calls.

2.2.1. Anomaly Detection Using System Call without Attributes

Forrest at al. [3] proposed to build an *n*-gram model of normal activity comprising possible sequences of system calls with *n* elements. To build such a model, they monitor process behavior in both synthetic and real environments. During the detection phase, the IDS evaluate the hamming distance between the current sequence of system calls and the normalcy model. Then, if the distance is greater than a specified threshold, the sequence is attributed to anomaly.

Durante, Pietro and Mancini [4] model the application behavior as a Finite State Machine (FSM), which accepts legitimate system call execution sessions caused by particular user commands. This approach requires comprehensive learning with expert who will trigger all possible commands of the application. If FSM does not accept the sequence of system calls invoked after given command, intrusion alarm will be triggered. This method is not applicable for a process which does not have user interface such as native services being most frequently chosen as targets for attacks.

Stolfo, Eskin and Lee [5] utilize call execution trees to derive a prediction model trough Sparse Markov Transducers. They also suggest using dynamic context-dependent window of contiguous system calls invoked by the process. During the detection phase they check the predictive (conditional) probability of the given subsequence against some threshold, then, if probability is less than the threshold, the subsequence is declared anomalous.

Anomaly-based systems which do not explore system call attributes usually show weak performance, since lots of critical information is discarded. For instance, network worm shell code may start command interpreter through "*CreateProcess*" function with inputs and outputs associated with a socket, what could be determined only by inspecting system call attributes. Moreover, without system call attribute data, it is impossible to relate system calls to some functional chains that is necessary to trace an OS objects manipulation session.

2.2.2. Anomaly Detection Using System Call Attributes

Liu and Martin [6] use system calls traces for detecting insider threats when a privileged user tries to perform a malicious activity. They utilize three different feature spaces to build a model of normalcy: *n*-grams of system call names, histograms of system call ID over fixed window and system calls with attributes. Each system call along with attributes is mapped onto its binary feature space so that dimensions are assigned to distinct values of every parameter. They use minimum hamming distance for *n*-gram models and system calls attributes models to detect anomalous records.

Tandon and Chan [7] use rule-learning algorithm to derive a model of benign behavior for every process. They take into account all arguments for individual system calls or bag of system calls (contiguous sequence) and build a set of specific rules. Feature vector for a single system call consists of such qualitative elements as: ID of system call, arguments, returned value and error status. Feature vector for a "bag of system calls" is composed of concatenated feature vectors of corresponding system calls from the bag. During the detection phase, feature vectors, inconsistent with the rules, are considered as anomalous with some degree of certainty.

Xu at al. [8], in contrast, analyze only critical system calls, which are vital for gaining access or control to the privileged target system. Their method also generates a pre-defined set of rules constituting the profile of normal behavior. Authors group system calls and assign level of danger to every system call. Nevertheless, they do not cohere system calls in functional chains that results in subjective information thus failing to perform reliable detection.

Kruegel at al. [9] suggests using arguments of the file management system calls which represent file names to be manipulated or accessed. Authors depict four models of normal strings of file name arguments: string length, character distribution, string structure and token structure. Having these four levels of model abstraction, the system based on this approach is able to detect malicious activity in terms of abnormal argument strings. Since the method is focused only on file manipulation it would not be able to detect “Executable Download and Execute” shell code in the case when the name of the worm image is consistent with the string models.

It is our observation that the limitation of *n*-gram and frequency-based models lies in specifics of input data. These methods disregard functional relationships between system calls and trace only contiguous system calls which may not be related by any functionality. This seems to be the main reason of high false positive rate reported by the authors. The problem with attribute models is that comprehensive learning may require large amount of learning space to contain all distinct records of models for all system calls for all processes. In addition, these models do not classify and group system calls to functional blocks, and the testing may result in high look-up time. In contrast, the success of the earlier mentioned DCA approach could be credited to the full utilization of attributes of system calls enabling the authors to reconstruct the “gene of self-replication” on a block-by-block basis [2].

2.2.3. Misuse Detection

Bernanshi, Gabrielli, Mancini [10] classify system calls with respect to the feasibility of utilizing individual system calls in compromising OS security and integrity. The IDS also contains the database of access control rules defined in the system call domain. Thus, the efficiency of the system depends on completeness of the rule base for a particular process. For instance, the IDS will block any attempt to run an executable which is never executed by the mediated process. While the authors mention the necessity of chaining system calls in order to reveal dangerous calls combinations, they do not provide any mechanism for assembling system calls into functional blocks. Hence, the system detects malicious activity only based on single system call misuse, what makes the security decision subjective and less reliable.

Kahg and Fuller [11] employ system call frequency distribution over fixed length window as in the input feature space and apply machine learning algorithms to make classification between malicious and normal system call traces. Again, the authors do not take into account high level functionality and system call semantics.

Sekar and Bowen [12] also developed high-level specification language to profile system calls usage with arguments for every process. Sequences of system calls which conflict with the rules of the language are treated as anomalous. While the authors trace sequences of calls with respect to attributes, they do not deduce explicitly OS object manipulation to reveal semantics of the system call chains what seems to be necessary to recognize a propagation engine.

In summary, some authors propose to retrieve quantitative information from system call data; others – to pre-classify system calls to explore categorical information. While such information does not provide enough knowledge to detect malicious activity with high confidence, there were a few attempts [10, 12] to deduce semantic information on the level of primitive functional blocks. However, recognizing merely primitive functional blocks would not provide complete picture of the process behavior.

We believe that without tracing the entire functionally of the shell code by restoring the complete procedure of OS object manipulation confident detection decisions cannot be made. To address the shortcomings of the referenced approaches, we propose to reconstruct the entire procedure of

OS object manipulation revealing particular high level operations and finally recognizing semantically expressed high level activity as a shell code algorithm. The proposed PED system implicitly conducts all the recognition steps while simulating the dedicated Colored Petri net.

2.3. Analysis of Propagation Engine Utilization

The largest library of exploits hosted by Metasploit Project [13] provides 18 possible shell codes for Windows OS, which could be attached to the exploit vector and potentially employed to compromise security of a susceptible host. Shell codes 1, 2, 3, 6 and 7 could be effectively utilized as a part of the propagation engine to achieve a worm proliferation into the target host. However, according to our observations, adversaries inherently employ limited set of particular types of propagation engines in the network worms.

To estimate the tendency of propagation engine utilization, we investigated 25 recent network worm families including: Sasser, Welchia, Blaster, Slammer and Mytob. Figure 6 depicts the propagation engine distribution among the studied worms. Worm propagation engines were determined based on Symantec virus database as well as reverse engineering particular strains of the worms. It could be observed that more than 60% of the worms employ “Bind shell” engine, while “Reverse shell” and “Executable Download and Execute” (ED&E) propagation engines are shared by 30% of the worms. Finally, less than 10% of the worms utilize other types of the engines such as thread injection, remote command execution and etc.

Table 1. Standard Shell Codes Available in Metasploit Project

1	Bind Shell
2	Reverse Shell
3	Bind DLL Inject
4	Bind Meterpreter DLL Inject
5	Bind VNC Server DLL Inject
6	Executable Download and Execute
7	Execute Command
8	Execute net user /ADD
9	PassiveX ActiveX Inject Meterpreter Payload
10	PassiveX ActiveX Inject VNC Server Payload
11	PassiveX ActiveX Injection Payload
12	Recv Tag Findsock Meterpreter
13	Recv Tag Findsock Shell
14	Recv Tag Findsock VNC Inject
15	Reverse DLL Inject
16	Reverse Meterpreter DLL Inject
17	Reverse Ordinal VNC Server Inject
18	Reverse VNC Server Inject

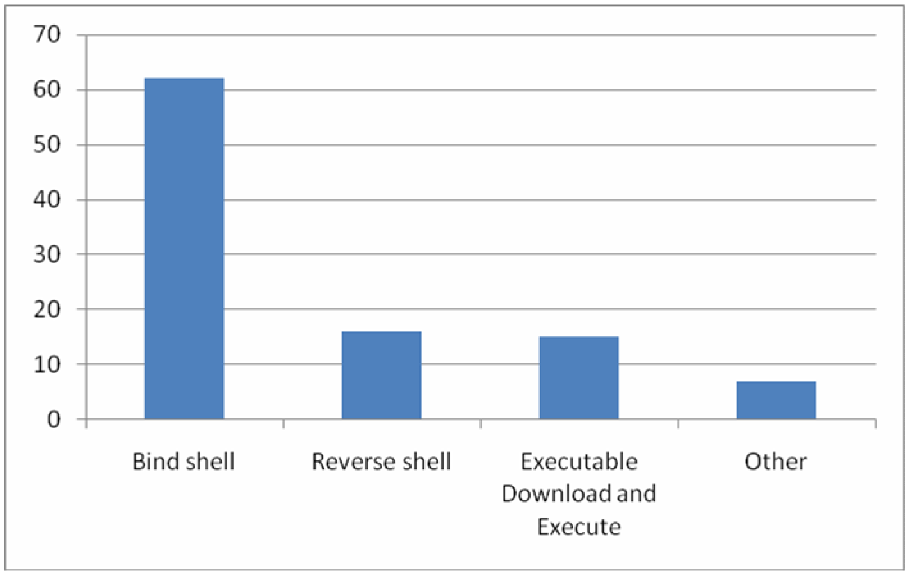


Figure 6. Propagation Engine Utilization

The first three engines (Fig. 6) are dominantly employed in the worms due to the simplicity of their utilization in the propagation session. Table 2 explains the high level operation of these engines. The first row of the table summarizes the functionality of the shell code of the propagation engines. The second row reviews the activity of the rest of the propagation engine which is not performed in the context of the exploited process, but by the legitimate means of OS.

One can see that the "Bind Shell" engine opens a network socket (port) and listens to the socket until attacker gets connected to the port. Then the connection is accepted and the shell code starts command interpreter, for instance "cmd.exe", whose input and output are associated with the socket. Eventually, the command interpreter process is set to listen for incoming commands and execute them. Previous three steps are performed by the exploited process while executing the shell code. The rest of the engine activity is the last step to achieve propagation. It could be seen that the attack source simply passes commands to make victim host to download worm image and run it.

The "Reverse Shell" engine is very similar to the "Bind Shell". However, in order to avoid inbound firewall, the shell code makes victim to connect to the attacker, instead of waiting for connection from the attacker.

Table 2. Propagation Engine Operation

	Bind shell	Reverse Shell	Executable download and Execute (ED&E)
Shell code (executed by the exploited process)	• Open a port • Accept connection • Create a command interpreter process that will listen on the port for incoming commands. (This will allow an attacker to issue remote commands on an infected computer).	• Open a port • Connect to the attacker • Create a command interpreter process that will listen on the port for incoming commands. (This will allow an attacker to issue remote commands on an infected computer).	• Connect to the server in the attacker host. • Download the worm image • Execute the worm image.
The rest of the engine (performed legitimately by OS)	Transmit commands to the victim host and make it to download worm image and execute it. For instance through TFTP.	Transmit commands to the victim host and make it to download worm image and execute it. For instance through TFTP.	

The ED&E engine performs the entire propagation in the shell code without post-activity as it is performed in first two engines. It simply creates a socket, establishes a connection to the attacker and retrieves a copy of the worm through the established channel. The shell code usually uses facility of high level protocols such as HTTP to download a worm, but sometimes it downloads the worm directly through the channel using merely TCP.

The above considerations indicate that a) different worms, "members" of different families tend to share the same propagation engines, and b) the number of totally different types of propagation engine is limited. At the first stage of the propagation session, the worm shell code is executed by the target, i.e. the vulnerable process. To achieve the propagation effect, the shell code has to utilize system resources trough utilizing API functions. As a result, each type of shell code would have its own system call execution pattern. Hence, one can detect and recognize particular system call signatures of the propagation engines. Therefore, the task of detecting worm attacks can be narrowed down to recognizing the propagation engines based upon the system calls signature.

2.4. Recognition of the Propagation Engine

As it was established above, during the first stage of propagation, the shell code invokes high level API functions. For instance, consider the operation of the shell code of the "Bind Shell" engine on the subsystem (API) level. It was noted above that the "Bind Shell" engine has quite primitive shell code to be executed by compromised process and this code invokes several high level APIs. One of the realizations of such a shell code for Windows OS is presented in Table 3. It could be seen that socket object is created using *Socket* API than it is put on listening state through *Listen* function and after accepting a connection the command interpreter ("cmd.exe") is started by *CreateFile* function with input and output set to the socket handle.

Table 3. Bind Shell Engine High Level Implementation

	API (function)	Parameters	Description
1	s=socket(…)	_out s	Opens a socket
2	bind(s)	_in *socket*=s	Bind the socket
3	listen(s)	_in *socket*=s	Put the socket to listen state
4	s1=accept(s)	_in *socket*=s _out s1	Accept a connection to the socket
5	CreateProcess("cmd.exe",…)	_in *pszImageName*="cmd.exe" _in *STARTUPINFO.hStdInput*=s1 _in *STARTUPINFO.hStdOutput*=s1	Start command interpreter with standard output and input being tied to the connected socket

A propagation engine may have several realizations with respect to high level functions. However, according to our experience, such high level variations tend to preserve the same implementation with respect to system calls. For instance, bind shell engine may be realized through anonymous pipes as well as through API's listed in Table 3. However, both high level realizations are translated into the same sequence of system calls. Hence, for the sake of reliability, propagation engines should be recognized at the system call level.

In order to recognize the shell code type, one has to model shell code activity in the system call domain. System calls by themselves do not provide a complete picture of the process activity; however, system calls create and manipulate objects by the means of handles and other system descriptors. Hence, we are more concerned of what shell code does with OS objects. For instance in the realization presented in Table 3, the "Bind Shell" engine creates an OS object named socket and puts it in the listening state to accept a connection. Thus, in the case of "Bind Shell" engine, we have to trace socket manipulation in order to detect the malicious functionality. To track object manipulation we have to relate system calls by object handles. In other words, the model has to follow inheritance of the object handles used by system calls and reveal chains of system calls. However, the shell code may have several chains not related to each other until some specific moment. Hence, the model must have the memory and the ability to trace system call chains in parallel. Therefore, it is required to provide parallelism, memory and inheritance, what could be addressed by utilizing Petri Nets [14].

Since system calls are related by the handle values, a Colored Petri Net (CP-net) must be utilized to formally describe the operation of a propagation engine in the system call domain. CP-nets are able to model particular activity in terms of actions (transitions) and states (places) at any desired level of abstraction thus providing the necessary generalization. Such a general model would be able to recognize the type of propagation engine in spite of possible code modifications or OS version. Moreover, due to the parallel structure of the Petri nets the model realization would be compact causing low computational overhead.

The CP-net could formally be presented as a tuple [14]:

$$CPN = (C, P, T, A, N, F, G, E, I) \qquad (1)$$

where: C – color set, P – set of places, T – set of transitions, A – set of arcs and etc.

The *color sets* constitute OS object handles employed by system calls and other system descriptors such as file names.

To address the specifics of our problem, we had to extend the original formalism of the CP-net in the way that *set of places* consists of tree disjoint dedicated subsets:

$$P = S \cup B \cup R\,, \tag{2}$$

where S – set of system calls places, B – set of functional blocks of system calls, R – recognition nodes.

Thus, each place of the set S corresponds to particular system call and its tokens are defined as ordered pairs of certain attributes of the executed system call. Places of the set B contain tokens representing successful execution of corresponding functional block. Recognition places from R represent successful recognition of the propagation engine.

Another extension we introduced into the classical CP-net is the inclusion of *inlet and outlet transitions*. The inlet transitions correspond to the system call execution which results in firing a new token to the corresponding place. The outlet transitions represent handle elimination, for instance through *NtClose* system call, which results in destroying token from the corresponding place usually belonging to the set S. Ordinary, transitions assemble system calls into chains (functional blocks) represented by places from the set B. Hence, each token represents an instance of execution of particular chain of system calls interconnected by object handles, pointers, etc.

In the context of the same process, different objects cannot share the same handle value. Hence the CP-net is free of conflicts what significantly simplifies implementation since we do not have to carry out conflict resolution policy.

Figure 7 shows a reduced version CP-net for the “Bind shell” propagation engine. The set of places $P = S \cup B \cup R$ of the full version of CP-net is defined in the following way:

$$S = \begin{Bmatrix} NtCreateProcess \mid NtCreateProcessEx, \\ NtCreateFile \mid NtOpenFile, \\ NtWriteFile, NtWriteVirtualMemory, \\ NtDeviceIOControl, NtClose \end{Bmatrix} \tag{3}$$

$$B = \begin{Bmatrix} "File\_Section", \\ "File\_Section\_Process", \\ "Socket\_open", \\ "Shell\_open" \end{Bmatrix} \tag{4}$$

$$R = \{"Bind\_Shell"\} \tag{5}$$

The network has a recognition node which represents an instance of “Bind Shell”. Due to space limitations, the Petri network is depicted without token outlet nodes which reflect object elimination through *NtClose* system call. Moreover, the network misses several alternative (undocumented) system calls which act the same as the original ones. For instance

NtCreateProcessEx is functionally identical to *NtCreateProcess*, but has different entry point address. The prototype IDS employs the full version of the Petri net with all necessary components included.

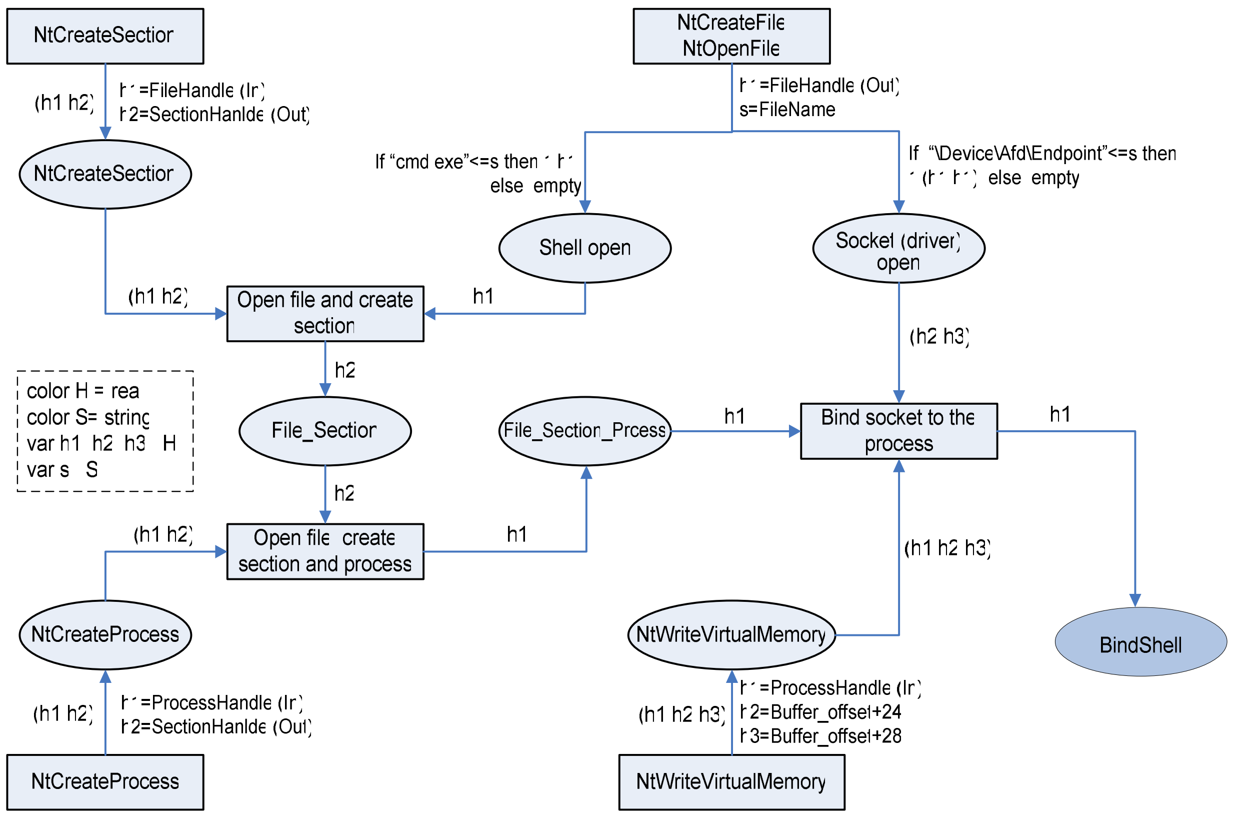


Figure 7. CP-net for Bind Shell Propagation Engine

The CP-net in Figure 7 has two color sets (types). The former type (H) is a handle and its variables represent object handles. The later color set (S) is a string which describes names of the files. Color and variable declarations listed in Figure 7 are written using the syntax of CPN markup language [14]. Also, the network has four inlet transitions: *NtCreateSection*, *NtCreateFile*, *NtCreateProcess* and *NtWriteVirtualMemory*. These transitions are enabled at the moment of successful execution of the corresponding system calls. The outgoing arcs of the system call transitions are provided with inscriptions which define token structure and variable initialization. For instance, when *NtCreateProcess* system call is invoked, the corresponding transition is enabled and fires a token which constitutes an ordered pair of two handles: handle of the process (input parameter) and the section handle (output parameter). There are also ordinary transitions such as "Open file and create section", which are enabled with particular events such as execution of the related chain of system calls. We used complex arc expressions to minimize the structure of the network. For instance, the expression of the ingoing arc to "Shell open" place checks if the file name contains the "cmd.exe" string, and if it does, the network fires file handle as a token to the place, otherwise it does not put anything.

The transition *NtCreateFile* is enabled if the corresponding system call has been executed. One can see that the token can be fired to one of the two places depending on the result of outgoing arc expressions with the *FileName* parameter. If the file name contains “\Device\Afd\Endpoint” string, then it means that the OS opens an instance of the AFD.sys driver which abstractly emulates a socket object. In fact, *Socket* function from *WS2_32.dll* invokes *CreateFile* function to open the driver instance and the handle of the driver will be ultimately employed as a handle of the socket object. Hence, in the Petri net a token pair of the handle is added to the “Socket open” place. This means that socket is created and is ready to be bound to a process.

The system calls *NtCreateSection*, *NtCreateFile* with “cmd.exe” and *NtCreateProcess* simply perform necessary steps for starting the “cmd.exe” process. We should point out, that only properly related system calls result in firing a transition. Hence, transition “Open file and create section” is enabled if *NtCreateSection* system call has file handle being equal to the handle of the opened “cmd.exe” file. Therefore, this transition fires section handle as a handle token to “File_Section” place only if ingoing tokens match each other (values of h1 variables are equal in both tokens). Place “File_Section_Process” corresponds to the state at which the address space of the process is allocated and Windows is ready to initiate the process. System call *NtWriteVirtualMemmory* maps *CreateProcess* parameters to the process address space. The transition “Bind socket to the process” is enabled if the input and output handles of the process are equal to the handle of the socket that indicates binding the opened socket to the process.

We intentionally omitted some minor system calls as transitions in the network. In particular, in the real network utilized in the prototype IDS, there is *NtDeviceIoControlFile* transition (system call) located after “Socket (driver) open place”. This system call sends commands to the driver ordering it to put the socket on *listen* or *accept* state. Moreover, we did not include *NtCreateThread* and *NtResumeThread* transitions (system calls) which constitute final steps in the process creation and running.

The CP-net depicted in Figure 7 is not merely a low level map of the high level implementation of the “Bind Shell” engine. Since, according to our experiments, different realizations and modifications of the original engine are detected by the same network, the Petri Net in Figure 7 is a general signature of the particular type of the propagation engine.

Furthermore, we designed CP-nets for the shell codes of such propagation engines as ED&E and “Reverse shell”. Parallel structure of Petri Nets allowed us to merge several networks of different engines into one general, multi-engine network. The simplified version of the general network is depicted in Figure 8. This network is able to recognize “Bind Shell”, “Reverse Shell” as well as ED&E propagation engines.

The multi-engine network has three recognition places which represent successful detection of the corresponding propagation engines. The part of the network which recognizes “Bind Shell” engine is structurally similar to the network depicted in Figure 7. However, the file name (variable *s*) is delivered to the place “Executable started” what allows to recognize the Bind Shell engine if the file name is “cmd.exe”, or the ED&E engine if the executed file has already been downloaded.

The network also contains two major states “Executable started” and “File downloaded”. “Executable started” corresponds to successful execution of *CreateProcess* API (kernel32.dll). “File download” could be achieved through the sequence of high level APIs (wininet.dll - InternetOpenURL, InternetReadFile, CreateFile, WriteFile), or through subsystem level implementation (kernel32.dll – Socket, Connect, Send, Recv, CreateFile, WriteFile). However, both high level realizations of “File download” functionality have the common implementation in system call level which involves *NtDeviceIOControlFile and NtWriteFile* system calls.

The general network is more compact due to the points of integration which are the parts related to *NtOpenFile* and “Executable started” place. The node “Executable started” is shared as an input by the recognition places of the corresponding engines. Hence, the entire substructure of the network (upper-left), which is involved in token delivery to the “Executable started” place, is shared and employed in recognizing of three separate engines. Such integration on the structural level significantly reduces size of the overall network.

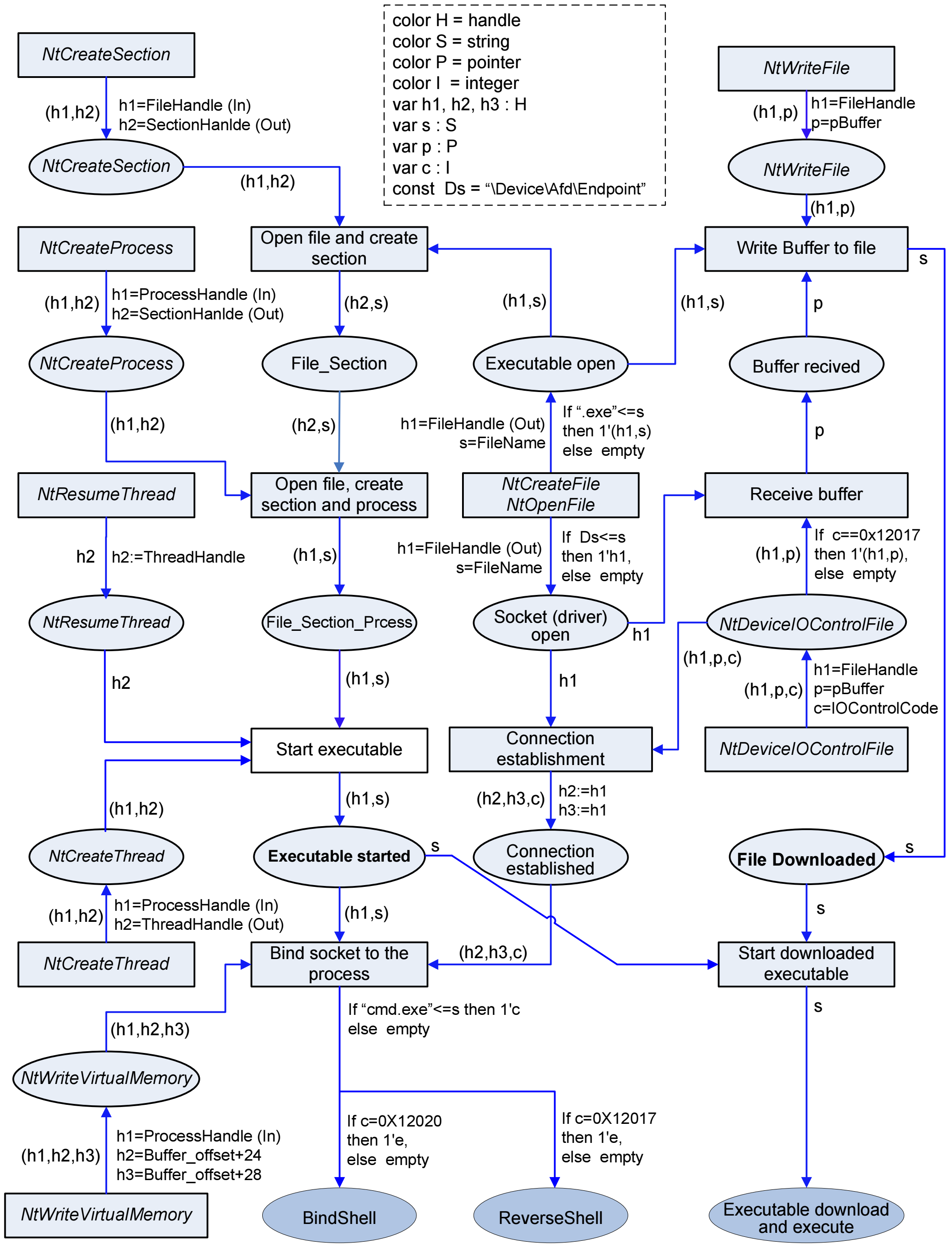


Figure 8. General "Multi-Engine" CP-net

2.5. Experimental Evaluation

In this section we evaluate the performance of the proposed approach. We designed a general CP-net capable of recognizing three propagation engines: ED&E, "Bind Shell" and "Reverse Shell". Then, we implemented a prototype of Propagation Engine Detector (PED) based on the designed CP-net and deployed a set of attacks in virtual environment to evaluate the efficiency of the proposed approach. Finally, we measured the CPU and memory overhead imposed by the PED system.

2.5.1. Experimental Setup

The experiments were conducted in the virtual network testbed at Binghamton University [15]. The testbed, being scalable up to 1000 nodes, was configured for virtual network comprising hundred victim hosts and one attacker host. The victim hosts were represented by virtual machines with vulnerable versions of Windows OS including the prototype PED. The attack was performed by a worm from the attacker host against each victim hosts resulting in worm propagation into the victim virtual machines.

We experimented with 10 notable network worms presented in Table 4. Every worm has been reverse-engineered deeply up to the point of revealing the exploit vector and shell code buffers supposed to be sent as payload in the attack packets. Some of the worms (Francette, Welchia) have shell code buffer encrypted and decrypt it just before sending. Hence, in order to extract exploit and shell code buffers we had to utilize run-time debugger and TCP dump software to record and process attacking packets payload. While reverse-engineering is not a trivial task, we have spent even more efforts to extract exploit and shell code of the propagation engines of the worms.

Having exploit and shell code buffers, we were able to incorporate them into a benign worm specially designed for the experiments. This experimental, completely observable and controllable worm allows for the development of user-specified attack scenarios. All realization of this worm equipped with various shell codes and exploits share the single payload which simply reports to the control server every worm propagation step: arrival, deployment, starting of the attack session and proliferation status. This worm also provides necessary functionalities for completing propagation session for different engines. Such functionalities include: TFTP/FTP server for “BindShell” engine and TCP server for uploading the image in case of utilization ED&E engine.

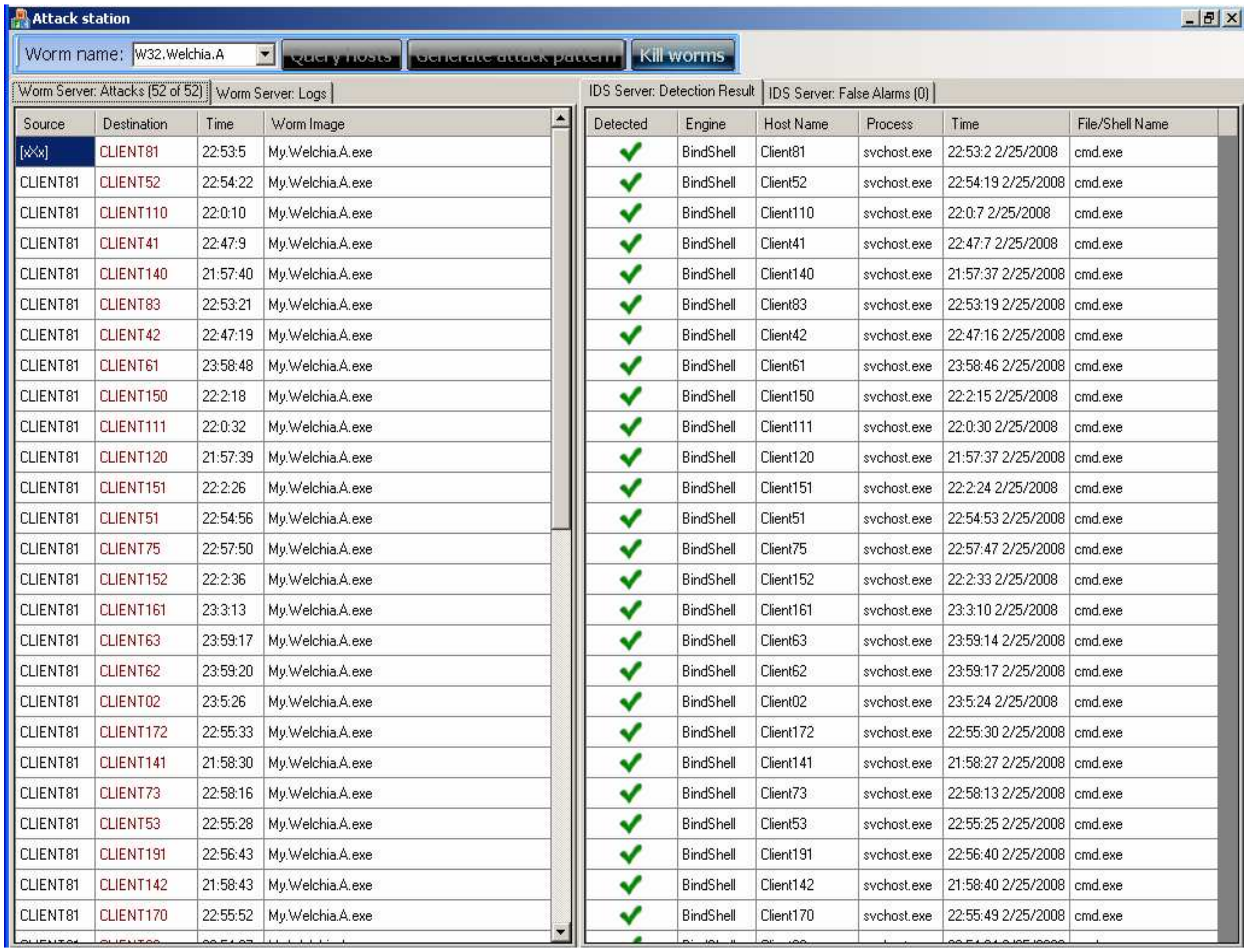
The utilization of the experimental worm with different shell codes of the originally tested “real” worms allowed for the test unification and assured the dependability of the results without sacrificing generality of the experiments. Moreover, many “real” worms have imperfections in certain modules utilized in propagation session, for instance W32.Welchia.A worm has bugs in TFTP server what may result in unsuccessful replication. However, the experimental worm has more reliable modules and its propagation rate could be much higher.

2.5.2. Results

We performed two sets of experiments. In the first set, we deployed attacks from the designated attacking host onto victim machines utilizing the experimental worm equipped with propagation engines of the “real” worms subjected to the investigation (Tab. 4). During these attacks the target hosts were in the idle mode, i.e. we did not perform or emulate user activity on the victim hosts during the attack session. Consequently, these experiments allowed for the evaluation of the rate of false negatives.

The propagation engine of every worm in Table 4 was employed in more than fifty consequent attacks perpetrated from the attacker host against randomly selected victim hosts. During these attacks our system monitored corresponding vulnerable processes and correctly detected and recognized propagation engines for every attack session showing no false negatives.

For the purpose of experiments we developed specific software (Attack station) that controls propagation of the experimental worm and collects alarm reports from local propagation detectors. Figure 9 shows a screenshot of Attack station at the moment when the custom worm with propagation engine of W32.Welchia.A completed infecting 52 virtual machines. In Figure 9, the left half of the table lists attacks reported by the worm instances just after being deployed in the victim machine. The attack report (generated by the payload of the experimental worm) includes the victim host name, attacker host name and time of starting the worm instance. The right half of the table to in Figure 9 (detection results) summarizes alarm reports received from the prototype PED systems.



Source	Destination	Time	Worm Image
[xXx]	CLIENT81	22:53:5	My.Welchia.A.exe
CLIENT81	CLIENT52	22:54:22	My.Welchia.A.exe
CLIENT81	CLIENT110	22:0:10	My.Welchia.A.exe
CLIENT81	CLIENT41	22:47:9	My.Welchia.A.exe
CLIENT81	CLIENT140	21:57:40	My.Welchia.A.exe
CLIENT81	CLIENT83	22:53:21	My.Welchia.A.exe
CLIENT81	CLIENT42	22:47:19	My.Welchia.A.exe
CLIENT81	CLIENT61	23:58:48	My.Welchia.A.exe
CLIENT81	CLIENT150	22:2:18	My.Welchia.A.exe
CLIENT81	CLIENT111	22:0:32	My.Welchia.A.exe
CLIENT81	CLIENT120	21:57:39	My.Welchia.A.exe
CLIENT81	CLIENT151	22:2:26	My.Welchia.A.exe
CLIENT81	CLIENT51	22:54:56	My.Welchia.A.exe
CLIENT81	CLIENT75	22:57:50	My.Welchia.A.exe
CLIENT81	CLIENT152	22:2:36	My.Welchia.A.exe
CLIENT81	CLIENT161	23:3:13	My.Welchia.A.exe
CLIENT81	CLIENT63	23:59:17	My.Welchia.A.exe
CLIENT81	CLIENT62	23:59:20	My.Welchia.A.exe
CLIENT81	CLIENT02	23:5:26	My.Welchia.A.exe
CLIENT81	CLIENT172	22:55:33	My.Welchia.A.exe
CLIENT81	CLIENT141	21:58:30	My.Welchia.A.exe
CLIENT81	CLIENT73	22:58:16	My.Welchia.A.exe
CLIENT81	CLIENT53	22:55:28	My.Welchia.A.exe
CLIENT81	CLIENT191	22:56:43	My.Welchia.A.exe
CLIENT81	CLIENT142	21:58:43	My.Welchia.A.exe
CLIENT81	CLIENT170	22:55:52	My.Welchia.A.exe

Detected	Engine	Host Name	Process	Time	File/Shell Name
✓	BindShell	Client81	svchost.exe	22:53:2 2/25/2008	cmd.exe
✓	BindShell	Client52	svchost.exe	22:54:19 2/25/2008	cmd.exe
✓	BindShell	Client110	svchost.exe	22:0:7 2/25/2008	cmd.exe
✓	BindShell	Client41	svchost.exe	22:47:7 2/25/2008	cmd.exe
✓	BindShell	Client140	svchost.exe	21:57:37 2/25/2008	cmd.exe
✓	BindShell	Client83	svchost.exe	22:53:19 2/25/2008	cmd.exe
✓	BindShell	Client42	svchost.exe	22:47:16 2/25/2008	cmd.exe
✓	BindShell	Client61	svchost.exe	23:58:46 2/25/2008	cmd.exe
✓	BindShell	Client150	svchost.exe	22:2:15 2/25/2008	cmd.exe
✓	BindShell	Client111	svchost.exe	22:0:30 2/25/2008	cmd.exe
✓	BindShell	Client120	svchost.exe	21:57:37 2/25/2008	cmd.exe
✓	BindShell	Client151	svchost.exe	22:2:24 2/25/2008	cmd.exe
✓	BindShell	Client51	svchost.exe	22:54:53 2/25/2008	cmd.exe
✓	BindShell	Client75	svchost.exe	22:57:47 2/25/2008	cmd.exe
✓	BindShell	Client152	svchost.exe	22:2:33 2/25/2008	cmd.exe
✓	BindShell	Client161	svchost.exe	23:3:10 2/25/2008	cmd.exe
✓	BindShell	Client63	svchost.exe	23:59:14 2/25/2008	cmd.exe
✓	BindShell	Client62	svchost.exe	23:59:17 2/25/2008	cmd.exe
✓	BindShell	Client02	svchost.exe	23:5:24 2/25/2008	cmd.exe
✓	BindShell	Client172	svchost.exe	22:55:30 2/25/2008	cmd.exe
✓	BindShell	Client141	svchost.exe	21:58:27 2/25/2008	cmd.exe
✓	BindShell	Client73	svchost.exe	22:58:13 2/25/2008	cmd.exe
✓	BindShell	Client53	svchost.exe	22:55:25 2/25/2008	cmd.exe
✓	BindShell	Client191	svchost.exe	22:56:40 2/25/2008	cmd.exe
✓	BindShell	Client142	svchost.exe	21:58:40 2/25/2008	cmd.exe
✓	BindShell	Client170	svchost.exe	22:55:49 2/25/2008	cmd.exe

Figure 9. Attack Session Perpetrated by W32.Welchia.A Worm

The alarm reports include: detected engine type, host and process name raised an alarm and time of detection. As it could be seen from the Figure 9, every attack was properly detected.

Moreover, the PED system triggers alarms several seconds earlier than the worm executes its payload that shows high efficiency of the proposed approach (the timelines of the detection decisions). Since the system detects particular functionality (shell code of propagation engine) before the worm actually replicates into the victim, we potentially can prevent worm proliferation without performing denial of service of the process being exploited. For instance in case of Bind Shell engine we can block any connections to the newly created command interpreter from the machine which connected most recently to the host that triggered alarm.

As it could be seen in Table 4, original worms utilize different exploits and some of them attack different Windows versions. However, we used the same version of PED in both Windows versions. The last column depicts propagation engine and a protocol used for retrieving the worm to the victim machine. We should point out that W32.Ibero worm and W32.Shelp worm utilize very different realizations of ED&E engine. W32.Ibero directly retrieves a worm copy through the open TCP channel, while W32.Shelp uses HTTP commands to download worm image from the dedicated web site. However, in spite of realization differences in engines, PED successfully recognized the propagation engine type what indicates high reliability of the proposed approach.

Table 4. Network Worms Being Tested

Worm name (aliases)	Vulnerability (MS code)	Target system	Propagation engine (upload protocol)
W32.Welchia.A	DCOM RPC (MS03-026)	Win XP Sp1	Reverse shell (TFTP)
W32.Sasser.C	LSASS (MS04-011)	Win 2000 Sp4, Win XP Sp1	Bindshell (FTP)
W32.Zotob.F (Bozori.A)	Plug and Play (MS05-036)	Win 2000 Sp4	Bindshell (TFTP)
W32.lberio (Hiberium.B)	Plug and Play (MS05-036)	Win 2000 Sp4	Executable download and Execute (direct download through TCP channel)
W32.Raleka	DCOM RPC (MS03-026)	Win XP Sp1	Bindshell (ECHO, direct injection)
W32/Alasrou-A (Small.D)	LSASS (MS04-011)	Win2000 Sp4, Win XP Sp1	Bindshell (TFTP)
W32.Kassbot(W32.Nanspy)	DCOM RPC (MS03-026)	Win XP Sp1	Bindshell (TFTP)
W32.Shelp	LSASS (MS04-011)	Win 2000 Sp4, Win XP Sp1	Executable download and Execute (HTTP from dedicated site)
W32.Blaster (Lovesan)	DCOM RPC (MS03-026)	WinXP Sp1	Bindshell (TFTP)
W32.Francette	DCOM RPC (MS03-026)	Win XP Sp1	Bindshell (TFTP)

The second set of experiments was performed without the deployment of attacks just to investigate false positives. The PED system monitored regular legitimate processes maintaining most windows services such as: lsass, svchost, winlogon, csrss and etc. PED also observed processes invoked by applications such as: acrobat, notepad, explorer, iexplore. On these virtual machines we browsed internet, manipulated files, managed various windows components and performed other activity trivial for an advanced user. None of the monitored processes caused false positive, since they did not exhibit any behavior being attributed to the shell code activity.

The above experiments showed zero false positive and zero false negative for a limited set of legitimate processes and ten worms (Tab. 4) tested during limited time period. While authors believe that the proposed approach is cable of detecting most of the existing worms and future worms utilizing standard propagation engines, such a high detection rate cannot be guaranteed for the future malware attacks. The authors do realize that no detection technique is perfect and it is expected that some sophisticated adversary may create a worm with conceptually new propagation engine which is not yet reflected by the CP-net of PED. However, any new propagation engine is expected to result in a certain pattern in system call domain that may be easily incorporated into the Petri Net of PED ensuring future detections of the worms based on the new engine.

2.5.3. Runtime Overhead

We utilized WinTask Professional (Uniblue System) to measure overhead of the detector. The PED system was executed in Windows XP Professional SP2 running on AMD Athlon 64 X2 (2200 Mhz) processor with 2 Gb of memory. Figure 10 shows CPU and memory resources consumed by PED in contrast to total usage over 10 minute interval. It could be seen that PED system utilizes less than 2% of CPU time in average with little spikes caused by high load of the entire system due to some intense tasks. Regarding memory usage, PED system consumes about 1.032 Mb of total 2Gb of memory.

Such a low CPU overhead could be credited to the fact that we hook and monitor only small set of system calls participating in particular CP-Net. Since the system call monitor is implemented in the Kernel mode, hooking small set of system calls spares large amount of computations needed to perform costly data transmission to PED system running in the user mode.

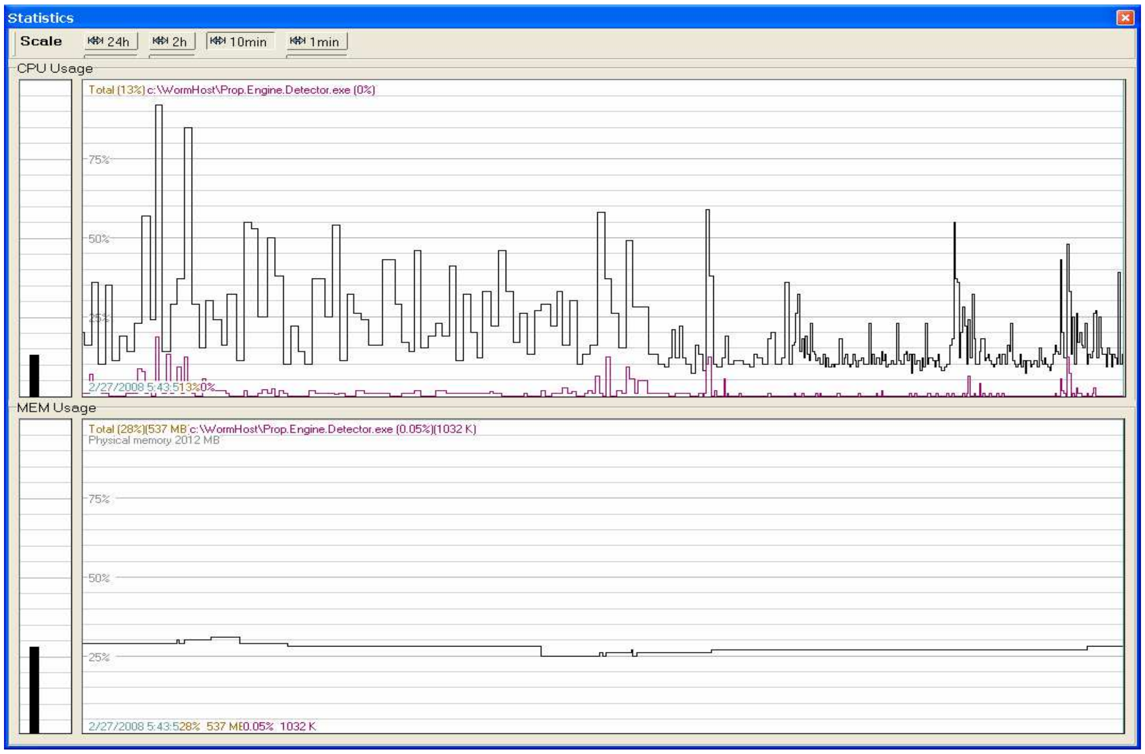


Figure 10. CPU and Memory Usage

2.5.4. Conclusion

A novel network worm detection approach based on identifying the shell code activity on the system call level which is an essential part of the propagation engine of the worm is presented. We employed Colored Petri Nets in Propagation Engine Detector (PED) system, which appeared to be very efficient in recognizing high level functionality in terms of OS object manipulation and attributing it to the malicious behavior.

Since we recognize propagation engines on the low (system call) level, different high-level realizations of the same type of propagation engine have been successfully identified with the same Petri net. Moreover, the PED system recognizes only the first stage of the propagation session (the shell code functionality), what makes the detector robust to various modifications of the original, standard propagation engines. Since polymorphism and metamorphism of worms/shell codes affects only binary structure of the code without affecting the system call execution pattern, the developed technology vulnerable to polymorphism and metamorphism of malware.

We developed a PED system and experimented with ten network worms and three propagation engines. The experimental results did not demonstrate false positives or false negatives showing high dependability of the proposed approach.

As a future work we propose the correlation of local alarms on network level in order to further increase the confidence of the detection decisions. Additionally, we plan to study possible ways to prevent consequent worm replication after the shell code has been executed and detected. Particularly, a set of engine-specific countermeasures timely blocking the worm propagation functionality could be developed. It is important that blocking functionality would not cause denial of service for operation system components.

3.0 HOST-BASED DETECTION: UTILIZING NON-STATIONARY MARKOV CHAINS

3.1. Introduction

The first Intrusion Detection System (IDS) utilizing system calls was proposed in [3]. Today, these systems utilize two main approaches, misuse detection and anomaly detection. Misuse or signature-based detection systems utilize descriptions of known attack expressed in terms of system calls. Although signature-based systems can provide high level of accuracy, they fail to detect previously unknown attacks. Anomaly detection systems utilize models of normal behavior of legitimate processes, especially privileged ones. These systems check the consistency between the invoked system calls and the profile of normality for a given process and have the potential to detect unknown attacks, though they frequently suffer from a high rate of false positives.

This research targets anomaly-based IDSs that in spite of their advantages are impractical due to high rate of false positives. The limited success of known research aimed at the alleviation of this problem [4, 7, 8, 16] in our view could be traced to the fact that it was primarily aimed at the improving the accuracy of the normality models (profiles) rather than achieving high confidence in classifying detected anomaly.

Two major contributions of this paper are as follows. Firstly, a novel host-level anomaly detection mechanism is proposed. Secondly, having efficient host level anomaly detection, we can declare a unique but rather simple principle, *false positives do not propagate*, that is suggested as the basis for establishing with high degree of confidence, whether detected anomaly is a false positive or a true positive.

In anomaly detection mechanism we utilize non-stationary Markov models. While many shell codes and exploits (in buffer overflow attack) may use only 20-30 system calls what would certainly be concealed in a histogram, Markov models are clearly preferable to other order insensitive techniques (such us frequency histograms) used to model normality profiles [3, 16]. However, the common assumption that the source (application or service) is a stationary stochastic process generally may not by true. Any application or services utilize high level functions intended to solve different tasks. When an application realizes several related tasks or group of tasks which condition each other, it is supposed to operate in one of its distinct phases (modes). For example, our preliminary experiments have identified the following major operation modes of Internet Explorer: application loading, browsing (loading pages from the Internet) and downloading (retrieving large files). These operation phases are distinguished by functionality and achieve different goals. Since different operation modes would have their own realization with respect to system calls, it can be assumed that operation phases have different unconditional as well as conditional distribution of system calls. Hence, the system calls profile of an application or service should be modeled as a non-stationary stochastic process.

We utilize so-called "moving omnibus" method to distinguish bifurcation points (points of sudden change in dynamics) in observed data, which would confine its stationary phases. Then we use obtained phases to train Markov models. As a result each process would have set of Markov models corresponding to each operation phase what would certainly increase model consistency.

We expect to dramatically decrease false positives by correlation of anomaly reports from different hosts in the network. The main distinguishable feature of viral software is self-replication. It is differently implemented in viruses and worms and can be revealed by the detection of specific (abnormal) sequences of system calls [2]. As the self-replication continues, the propagation of the same abnormal activity pattern could be observed within the network. We call it the anomaly propagation.

The utilization of system call attributes provides unambiguous representation of the connectivity between various computers and processes within the network. Then, if the anomaly propagation pattern is consistent with the process connectivity pattern, it could be declared with a high degree of certainty that the detected anomaly is *true positive*, otherwise it is *false positive*.

The proposed IDS approach has two levels of implementation, the host-level anomaly detection, and the network-level attack detection. We propose a new concept targeted to decrease false positive of anomaly based IDS in system call domain. To mitigate false positives, we suggest network based correlation of collected anomalies from different hosts as well as new means of host-based anomaly detection.

We formulated a novel concept of anomaly propagation that could be summarized as: false alarms must not propagate within the network. Therefore, should an anomaly propagation be observed, we can attribute it to a propagating worm, otherwise the alarms are to be treated as false positives. The rationale behind the concept lies in the fact that the most common feature of worms and viruses is self-replication. As replication takes place, a malicious code propagating through the network would carry out the same activity resulting in almost identical system call sequences and triggering the same alarm at different hosts. While system calls clearly reveal interactions between hosts, the alarm propagation effect would be detected what distinguishes "true alarms" from "false positives".

Efficiency of propagation analysis presumably depends on accuracy of the anomaly detection on the hosts. Existing host based anomaly detection schemes may not be dependable enough to provide solid basis for new anomaly correlation concept. Therefore, to make propagation analysis feasible, we propose a new anomaly detection mechanism operating on the host level that employs non-stationary Markov models. Many applications or services may operate in different modes, which have different dynamics with respect to system call issuance modeling. Therefore, we treat application or service as a non-stationary stochastic process and model it as a non-stationary Markov chain what significantly improve model consistency compared to stationary Markov chain.

3.1.1. Related Work

Signature-based IDS utilizing system call data are known in literature. The feasibility of anomaly detection using system calls is shown in [3], [4], [16]. The efficiency of this approach can be enhanced further by the analysis of system call attributes as shown in [6], [7], [8], [9], and [17]. Additional improvement of this approach was demonstrated in [2]. The misuse detection-based IDS approach could be best exemplified by [18], [19].

3.2. Anomaly Detection

Traditionally anomaly detection procedure consists in recognizing process behavior deviation from the profile of normalcy. On the system calls domain the IDS looks for abnormal trace of system calls according to the model and consider such trace as anomalous. The efficiency of the anomaly detection depends on the model accuracy. The best way to model the application in the system calls domain is to derive system call execution graph which would explicitly reflect all possible braches in algorithms. However, it is impossible to process all possible branches of initial algorithms from the binary code due to the implicit logic transitions (jumps). Thus we can only derive approximate model, what adds some degree of uncertainty in the application description. To reflect such uncertainty, we can assume that system calls emitted by some stochastic source (an application) with categorical state space represented by system calls.

Any application or services utilize high level functions these functions are intended to solve different tasks. When an application realizes several correlated tasks or group of tasks which condition each other, it is supposed to operate in one of its distinct phases (modes). These operation phases are distinguished by functionality and achieve different goals. Since different operation modes would have their own realization with respect to system calls, it can be assumed that operation phases would have different unconditional as well as conditional distribution of system calls. Hence, system calls profile of an application or service can not be modeled as stationary stochastic process, but as non-stationary stochastic process.

Operation phases consist of many system calls and implement some strictly prescribed high-level tasks. This consideration assumes the source to be stationary over each operation phase. Since dynamics of the stochastic process are appeared to be invariant over an operation phase, we can model operation phases by Markov chains. Therefore, the source would be modeled by set of Markov models corresponding to each operation phase.

In this context, the trace of system calls is considered to be anomalous if it is not likely to happen according to current Markov model (corresponding to current operation phase). The sequence is not expected to happen if it was not predicted by Markov model. Therefore, the anomaly score can be chosen as prediction performance of the Markov model over the observed sequence. Prediction performance can be represented by chi-square likelihood ratio of the observed sequence of certain window. If chi-square likelihood ratio exceeds specified threshold we declare observed sequence as anomalous.

3.3. 3.3 Operation Phase Detection

Before deriving Markov models, operation phases must be distinguished automatically in unsupervised fashion. We have to apply such method which for the given sequence of observations (system calls) would determine bifurcation points (moments of dramatic change in process dynamics). These bifurcation points would certainly correspond to moments of operation phase switching.

One of the most efficient techniques for detecting bifurcation points is moving "omnibus" method. The method is simple extension of rather classical "omnibus" method. The latter one uses Pearson's χ^2 hypothesis test. The observed sequence of states (system calls)

$$(S = \{s_1,...,s_n\}) \qquad (6)$$

is partitioned by two contiguous subsequences ($S_1 = \{s_1,...,s_k\}$ *and* $S_2 = \{s_{k+1},...,s_n\}$) for some dividing point k. Then, every subsequence (S_1, S_2) is used as training set to compute corresponding transition matrix (T_1, T_2). We can state the following test to check if *k* is bifurcation point:

$$H_0 : T_{1,2}(i,j) = T(i,j), \quad versus \quad H_1 : T_{1,2}(i,j) \neq T(i,j) \qquad (7)$$

where, *T* is global transition matrix computed over the entire sequence *S*.

If the null hypothesis is rejected then transitional probabilities are time variant due to dynamic change in point *k*. Therefore, rejection of H_0 points out that *k* is bifurcation point with some degree of confidence. To implement the test (7) we can compute test statistic in the following way:

$$W = \sum_{k=1}^{2} \sum_{i,j=1}^{m} \left(N_1(i) \frac{(T_1(i,j) - T(i,j))^2}{T(i,j)} + N_2(i) \frac{(T_2(i,j) - T(i,j))^2}{T(i,j)} \right) \qquad (8)$$

where $N_1(i), N_2(i)$ - marginal observed frequency of i-th state derived form first and second subsequences respectively.

By the central limit theorem, statistic W is asymptotically distributed as χ^2 with $2(m-1)^2$ degree of freedom under null hypothesis. Hence, we can compute p-value of the test in the following way:

$$p_{value} = 1 - F_\chi(W) \qquad (9)$$

where $F_\chi - \chi^2 \, cdf$. If p-value lower than chosen test size α, we reject H_0 and claim non-stationarity with $|\alpha - p_{value}|$ significance.

The “moving omnibus” method consists in deriving p-value for test (1) with respect to center point of the window sliding over the observed sequence (trace). The sliding window must be long enough to derive local Markov models from the left and right halves of the window. The points rejecting null hypothesis would be declared as bifurcation points. Having set of bifurcation points we can chose the most appropriate ones according to constraints for phase minimum length and number of phases. The algorithm for selecting such points is proposed below.

Input constraints:

n –number of locally stationary phases

$l_{\min}$ – minimum length of a phase

Algorithm:

1. Form list *L* of sorted bifurcation points in decreasing order with respect to significance.

2. Take first n points from list L and compute length of phases enclosed by these points

3. For every phase, which length is less than $l_{\min}$ delete from the list L less significant boundary point.

4. Continue step 2 until all constrained are met or there is no enough points left in the list.

If the process has enough locally stationary periods, the algorithm will determine bifurcation points enclosing these periods in the observed sequence.

Let us demonstrate efficiency of the algorithm on real process. The sequence of system calls was observed during two operation phases of the Internet Explorer. First phase was browsing through different sites without downloading big files and the second phase was downloading big files from some sites. It was known that system calls following 15000 were invoked during downloading phase. Figure 11 depicts results of the algorithm. Plot shows p-value for every separated point moved from 5000 instance to the end of the sequence (40000). The size of hypothesis testing was 5 percent. It could be seen that the test was rejected only once in the separating point 15000 what shows high accuracy of bifurcation point detection. P-value changed dramatically in earlier points (13000-14000), but never reached significance level. The results of the experiment showed high efficiently of the moving “omnibus” method.

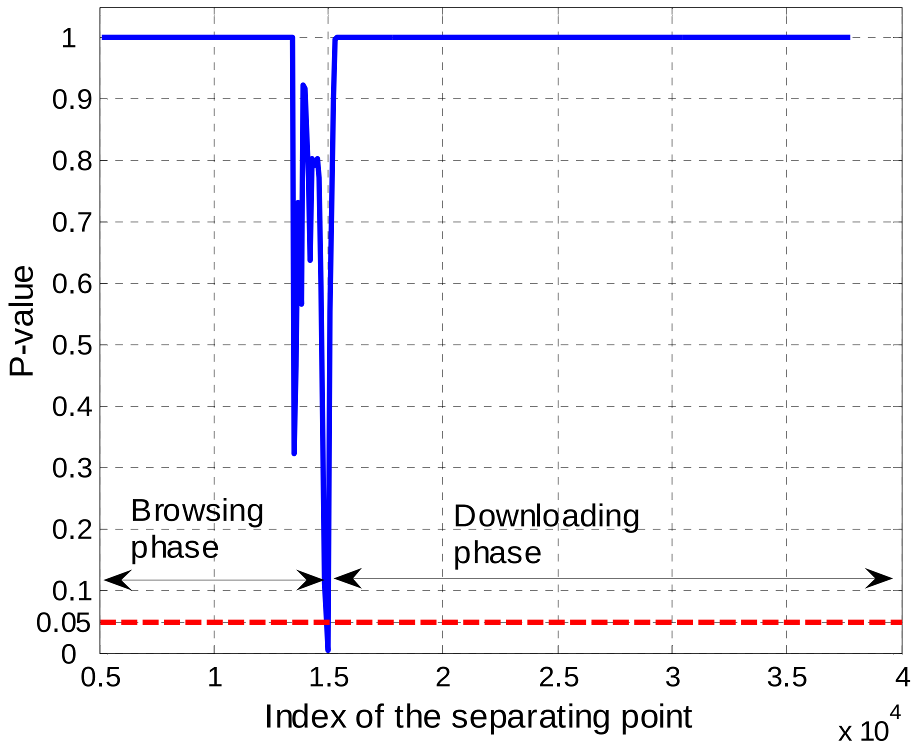


Figure 11. Bifurcation Point Detection

Having facility to recognize stationary operation phases in observed sequence of system calls, we can use these segments (subsequences) of states (system calls) to derive Markov models in offline. Nevertheless, in testing regime, it is necessary to recognize (in real time) what phase the monitored application operates in to apply corresponding Markov model. The problem of matching current outcomes to set of models was addressed in several publications (e.g. Stolfo [16]). Many authors use simple approach consisting in trying different models form the set and choosing the one which best fits according to some distance metric. We used the similar method, if current Markov model is not consistent according to predicting performance metric (likelihood ratio), the system searches for the model having the greatest performance. To avoid undesired frequent model change we introduced a constraint on minimum number of system calls before model switching is allowed.

3.4. Markov Models Order Detection

System calls are invoked according to some algorithm which has a specific logical structure and solves its own tasks which constitute operation phases. Since system calls are issued in consecutive logic order, the probability of occurrence of system calls depends on not only one previous system call, but several preceding system calls (prefix of system calls). These considerations lead to necessity of using high order Markov models what is expected to further increase the models efficiency with respect to [19]

The order of the Markov chain corresponding to an operation phase is unknown, hence has to be determined. There are several approaches to determine order of Markov model consistent with observed sequence [20]. These methods include: transition correlation based method, chi-square statistics of the transition frequency, index of transition complexity and information theoretic approach.

In the proposed system, information theoretic approach is used to determine proper order of Markov model. The rationale beside the approach is the fact that if probability of occurrence of a state highly depends on *n* previous states then mutual information of the current state and n preceding states must be greater than information based on less than *n* past states. The criteria for defining best order of the Markov model can be formally presented in the following way:

1. Starting from $n = 1$

    5. Compute n-gram transition probabilities: $p(i \mid \underline{i_1,...,i_n})$

where, $p(i| \underline{i_1,...,i_n})$ probability of occurring state i given past prefix of states $i_1,...,i_n$ (order preserving prefix)

    6. Compute *n+1* and *n* order mutual information:

$$I(X_{n+1}; X_1,...,X_n) = \sum_{i_1,..,i_{n+1} \in 1..N} p(\underline{i_1,..,i_{1+n}}) \log\left(\frac{p(\underline{i_1,..,i_{1+n}})}{p(i_{n+1})\, p(\underline{i_1,...,i_n})} \right) \qquad (10)$$

7. $I(X_n; X_1,..., X_{n-1})$ - to be computed analogically

$$D= I(X_{n+1}; X_1,..., X_n) - I(X_n; X_1,..., X_{n-1}) \quad (11)$$

8. If $D < \tau$ then stop, otherwise $n = n+1$ and go to step 2

This algorithm will derive the maximum order of the model which still provides specified mutual information increase. In reality mutual information rate is decreasing function of the model order for given observation sequence. Our preliminary experiments showed that phases have maximally third order models.

3.5. Anomaly Detection

The abnormal sequence of system calls will have the low likelihood ratio what is indicative of unpredictability by the Markov model. We employ the log likelihood ratio:

$$L(\mathbf{X}) = -\sum_{i=1}^{|\mathbf{X}|} \ln(p(X_i \mid X_{i-1}X_{i-2}...X_{i-k})), \quad (12)$$

where $\mathbf{X}$ is sequence of the system calls in question, k - Markov model order. Note that the higher the $L(\mathbf{X})$, the less predictable is the sequence.

Abnormal sequence of system calls must result in high peak of the log likelihood ratio which value is bigger than some threshold. Then the system calls representing the points of the segment higher than threshold would be recognized as abnormal sequence. However, according to our experience, the shell code may have inconsistent likelihood ratio due to some high level APIs which are used frequently during legitimate activity of the process and consequently reflected in the model. For instance in the Figure 12, the dashed line is likelihood ratio and the entire segment (marked by arrow) corresponding enclosed by two high blue spikes should be treated as anomaly. However, one can see that the central part of the segment (marked by rectangle) has log likelihood less than threshold. This will partition the sequence and eventually decrease the efficiency of the system.

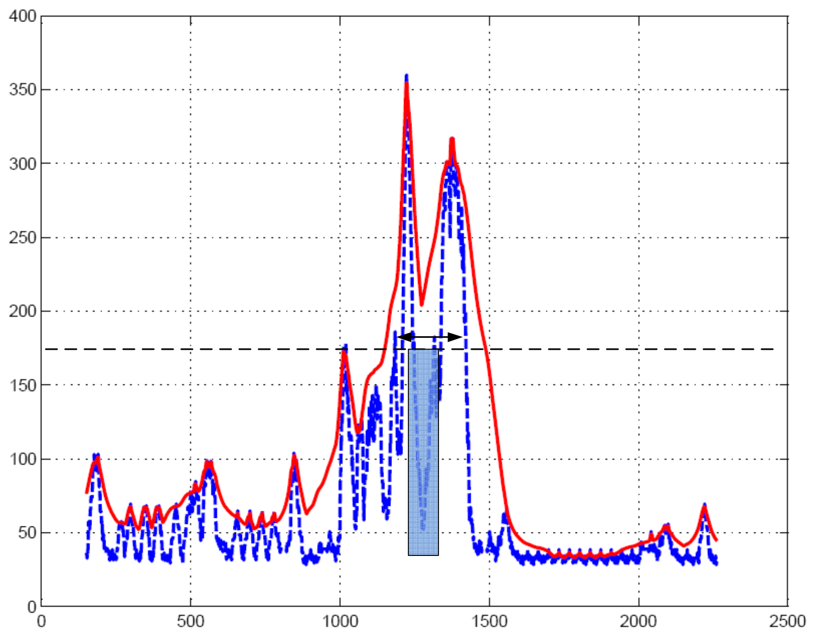


Figure 12. Transitivity Closure of the Log Likelihood Ratio

To address such shortcomings we proposed to utilize transitivity closure as maximum mean value of sub-segments of the original segment. Hence the new function would be expressed in the following way:

$$\hat{L}(i) = \max_{j,k<\tau}\left(mean\left[L(i-j)...L(i+k)\right]\right) \quad (13)$$

It could be seen that the new function solid line has consistently high value over the likelihood spikes and does not have rapid drop allowing detecting the abnormal sequence of system calls as a permanent segment.

System calls are invoked according some algorithms which have logical structure and solves its own tasks which constitute operation phases. Since system calls are issued in consecutive logic order, the probability of occurrence of system calls depends on not only one previous system call, but several preceding system calls (prefix of system calls). These considerations lead to necessity of using high order Markov models.

The order of the Markov chain corresponding to an operation phase is not known, hence has to be determined. There are several approaches to determine order of Markov model consistent with observed sequence. These methods include: transition correlation based method, chi-square statistics of the transition frequency, index of transition complexity and information theoretic approach.

We used information theoretic approach to determine proper order of Markov model. The rationale beside the approach is the fact that if probability of occurrence of a state highly depends on *n* previous states then mutual information of the current state and n preceding states must be greater than information based on less than *n* past states. The criteria for defining best order of the Markov model can be formally presented in the following way:

1. Starting from $n = 1$

    9. Compute n-gram transition probabilities: $p(i \mid \underline{i_1,...,i_n})$

    where, $p(i, \underline{i_1,...,i_n})$ probability of occurring state i given past prefix of states $i_1,...,i_n$ (order preserving prefix)

    10. Compute *n+1* and *n* order mutual information:

$$I(X_{n+1}; X_1,...,X_n) = \sum_{i_1,...,i_{n+1} \in 1..N} p(i_{n+1},..,i_1) \log\left(\frac{p(i_{n+1},..,i_1)}{p(i_{n+1})\, p(i_n,...,i_1)}\right) \qquad (14)$$

    11. $I(X_n; X_1,...,X_{n-1})$ *- to be computed analogically*

$$D = I(X_{n+1}; X_1,...,X_n) - I(X_n; X_1,...,X_{n-1}) \qquad (15)$$

    12. If $D < \tau$ then stop, otherwise $n = n + 1$ and go to step 2

This algorithm will derive the maximum order of the model which still provides specified mutual information increase. In reality mutual information rate is decreasing function of the model order for given observation sequence. Our preliminary experiments showed that phases have at most third order models.

3.6. Experimental Results

We performed trace based simulation using recorded system calls were from legitimate processes as well as malicious software. As malicious agent, we choose forth generation of Sasser worm – Sasser.D worm. Simulated legitimate processes include Microsoft Internet Explorer and CCAPP. We do not claim comprehensive monitoring in these preliminary experiments, however size of records constituted tens of thousands. For instance, we used 50000 system calls issued by Internet Explorer for crating Markov models. We also recorded 24 contiguous system calls invoked by victim process executing Sasser worm payload.

Using the system call records, we obtained non-stationary models for three chosen processes. Figure 13 depicts call prediction performance for CCAPP process based on stationary Markov model versus non-stationary model. Non-stationary model contains three dynamic invariant chains. The trends present chi square likelihood ratio statistic which formally reflects prediction performance, the lower statistic the better prediction. Examining the curves we can see that non-stationary Markov model (solid line) totally outperforms stationary model (dashed line).

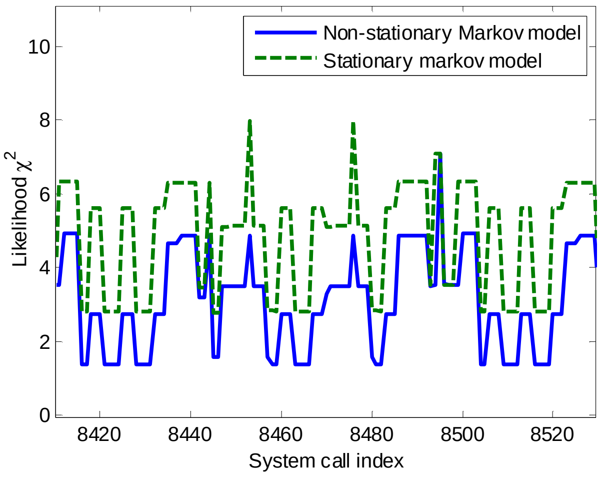


Figure 13. Predicting Performance for CCAPP

4.0 SERVER-LEVEL INTRUSION DETECTION

4.1. Anomaly Replication Detection

The main distinguishable feature of an activity perpetrated by malicious software such as worms or viruses is self-replication. Viruses have to create a file containing its copy or attach itself to some victim file. In contrast, worms do not have to undertake any file operations in the memory space of the victim process, but perform some activity on behalf of the victim, legitimate process. Hence, worms may not leave any file “traces” which can be used to detect self-replication. Worms attack packets consisting of an exploit vector followed by a propagation engine. To perform some activity the worm code must utilize system resources through system calls, which will be certainly reflected in the process behavior and will be detected as an anomaly.

To the best of our knowledge, modern worms could be multi-exploit and packet polymorphic, however they are expected to utilize the same propagation engine in every attack. The biggest collection of exploit payloads offered by the Metasploit Project [13] has several propagation engines including: bind shell, thread injection, windows remote control, etc. For instance, for bind shell, the propagation engine runs the command shell having input bound to a socket being in a listening mode. Usually, adversaries utilize one propagation engine in their worms. As a result, the same functionality would be carried out at every instance of an attack. Attacks with several propagation engines are possible as well and are addressed later.

One can see the advantage of tracing the system calls instead of the packet contents. While an adversary may write a polymorphic worm whose attack packet payload may be different from instance to instance, every attack would carry out a functionally invariant propagation engine which leads to the same system calls execution pattern. For instance, in the case of a bind shell payload, some instructions can be changed preserving functionality of the engine, but still the payload has to create a socket, assign it to a port, create a command shell and accept a connection, which requires invoking a certain sequence of system calls. Hence, every copy of the worm would carry out the same activity that would result in similar system calls traces and similar detected anomaly for the victim processes. Therefore, the self-propagation could be detected in terms of anomalous system calls traces reoccurring on different hosts of the network. We call this phenomenon “anomaly propagation”.

To distinguish the “anomaly propagation” from a set of coincidences (meaningless occurrence of the same anomaly at different nodes of the network), one has to analyze the connectivity pattern of the nodes. If the propagation pattern is consistent with the connectivity pattern between hosts, we claim that “anomaly propagation” takes place due to malicious activity. On the other hand, if anomaly propagation is inconsistent with the connectivity pattern, we declare it be a false positive.

Anomaly propagation analysis requires distinguishing similar or equivalent abnormalities indicative of the same propagation engine. Anomalous segments of system calls detected on the hosts are separated into thread specific sequences, reported to the server, and provided with a time stamp and source host ID. Detectors must split abnormal segments into thread specific sequences because the worm shell code is executed by one thread.

A server compiles the received anomalies into different groups containing equivalent system call sequences. The distance between two anomalies is defined in the following way:

$$\underline{d}(S_1,S_2)=1-\frac{\max\left(\left|C(S_1,S_2)\right| \mid \left(\left|C_m\right|>k\right)\right)}{\min(|S_1|,|S_2|)} \qquad (16)$$

where $\left|C(S_1,S_2)\right|$ is the length of common subsequence of anomalous strings and C_m is the minimal common factor of the compact set of factors representing the subsequence. The numerator specifies the value of k as the minimum length of the common segments of two sequences (S_1,S_2) declared to be equivalent.

Initially, we used just the length of the longest common segment, but eventually we realized that this definition would be significant only for propagation engines having a linear execution path. If a propagation code has some branches with the execution path, then the common factor may be fragmented and some segments may diverge for two anomalies recorded from different attack instances. The distance (16) tolerates the existence of relatively small call execution branches within the matching segments. In fact, the metric (16) is not really a distance, since it does not satisfy triangular inequality; however, it seems to be a suitable measure of anomaly similarity.

An anomaly equivalence measure can be defined with respect to distance (16) as well as another metric:

$$\bar{d}(S_1,S_2)=1-\frac{\max\left(\left|C(S_1,S_2)\right| \mid \left(\left|C_m\right|>k\right)\right)}{\max(|S_1|,|S_2|)} \qquad (17)$$

One can see that (17) and compute the percentage of the non-common part for shorter and longer system call strings respectively. Distances (17) and provide the basis for the establishment of specific groups of equivalent anomalies defined in terms of system calls.

During the execution, a reported anomalous string (sequence) would first be checked against the existing groups and will be added to the appropriate group as an equivalent to the anomalies contained in the group. Formally, anomaly S is added to the group G that contains string R if the following equivalency relation φ holds:

$$S\varphi S_i \in G \ \ iff \ \ \left(\underline{d}(S,R)<\tau_1\right)\&\left(\bar{d}(S,R)<\tau_2\right) \qquad (18)$$

One can see that the equivalency relationship (18) is reflexive, commutative, and transitive for group members. Hence, group members are formally equivalent according to the relationship (18).

If none of the groups is equivalent to the new anomaly, the system will continue checking the anomaly with the anomalies contained in the pool looking for a candidate for the new group. We should point out that, an anomaly cannot be added to the group if the group already contains an anomaly reported by the same host. To avoid pool inflation, close anomalies from the same host are represented by the longest of them.

The system tracks the group size (number of members) and if the size exceeds some threshold, the system analyzes the pattern of propagation of anomalies in the group to reveal the fact of replication. Replication activity is detected using the connectivity pattern (graph) that is represented by weighted directed graph. Such a graph $C(V,E)$ with nodes being hosts and edges connecting attacker and victim hosts weighted with relative time of the TCP session beginning in case of TCP port and UDP packet arriving in case of UDP port to be listened by the process issued anomaly.

Anomaly propagation is considered to contain a replication pattern if it is consistent with the connectivity graph in both topological and time sense. In other words, if anomaly is replicating, it must propagate according to simple rules:

- Each new instance of anomaly (except the first one) must occur in the process which has recently been interacted by another suspicious process (which already reported an anomaly).
- The time elapsed from the last interaction and anomaly occurrence must not be longer than the prescribed threshold (active window).

For the multipartite attack (coordinated multi source malicious activity) the first rule must tolerate the presence of several sources.

Iterative algorithm verifying if anomaly propagation in the group has replication pattern is straightforward:

Input information:

- New anomaly V_A being added to the group $G = \{V_1,..,V_n\}$
- Weighted adjacency matrix T of the connectivity graph $G, T(i,j) = weight(E(V_i,V_j))$

Algorithm:

1. Get the subset of group members connected to anomalous process:

$$S = \{V : (V \in G) \& (T(V,V_A) > 0)\} \tag{19}$$

2. Check if the last connection time is less than threshold: $\min\limits_{V \in S}(T(V,V_A) < t_{\max}$

3. Increment the counter of the group members participating in the replication $k = k+1$

4. If the normalized counter is exceeds the threshold ($k/|G| > \tau$), declare that the network is under attack

5. If the group size exceeds the prescribed value and the normalized counter value is under the tolerance value $(k/|G| < tol) \& (|G| > s_{\max})$, declare a false positive.

The rationale behind the proposed distance formalism lies in the fact that processes may be attacked and subverted at any time. Thus we can expect that anomaly sequences would have different short suffixes (segment of legitimate system calls). Subsequence from the beginning to the legitimate suffix would be a large segment forming the pure sequence of system calls actually invoked by the worm's payload. Hence, anomalous sequences caused by the same worm will have large common factor which would constitute some segment of "malicious" system calls.

The longest common factor between two strings (with length n, m) can be found through dynamic programming with $O(n \cdot m)$ computational complexity. Hence, determining the closest group by a one-to-one search would exhibit $O(N \cdot m)$ complexity, where N – is the sum of the length of strings in the cumulative anomaly dictionary. However, we do not need to know the longest common factor itself to compute the distance $d(S,G)$, but only the length of the factor. The length of the $c(S,\tilde{S})$ can be approximated by the length of longest common subsequence which can be found trough weighted Levenshtein distance for the case when $c(S,\tilde{S})$ constitutes significant part in both sequences. The longest common subsequence is not necessary contiguous, but due to high performance of the proposed anomaly detector, one can expect that the "alien" part of the anomalous substring (common factor) would be much longer than the legitimate one thus justifying the approximation.

The edit distance is weighted so that, deletion operations would not constitute any penalty score. Formally, the length of the longest common factor of sequences S and $\tilde{S}$ can be represented in the following way:

$$\begin{cases} \left|c(S,\tilde{S})\right| = |S| - L_W(S,\tilde{S}) \\ L_W(S,\tilde{S}) = L(S,\tilde{S}) - L_{delete} \\ L(S,\tilde{S}) = L_{sub} + L_{insert} + L_{delete} \end{cases} \qquad (20)$$

where, $L(S,\tilde{S})$ - is the Levinstein (edit) distance between S and $\tilde{S}$, $L_{sub}, L_{insert}, L_{delete}$ - are number of substitution, insertion and deletion operations respectively, and $L_W(S,\tilde{S}) = L_{sub} + L_{insert}$ represents the weighted edit distance with zero penalty of deletion operation.

Expressions (17) and (18) show that by minimizing $L_W(S,\tilde{S})$ we minimize the distance $d(S,G)$. The problem of finding the closest anomaly group can be reformulated in the following way: given the dictionary of strings (sequences) D and a pattern string S, find the string $\hat{S}$ from the dictionary which is the closest to S with respect to weighted edit distance $L_W(S,\tilde{S})$. Having the closest string $\hat{S}$, we consider the distance (1) as distance to the group the string $\hat{S}$ belongs to.

Such a problem is called *approximate dictionary querying* and is addressed in several papers [13, 15, 20]. Yates and Navarro [20] use metric property of edit distance (triangular inequality) to neatly organize vocabulary as a metric space. Such data structure reduces dictionary query complexity to $O(N \log N)$. Brodal and Gasieniec utilized cell-probe model to achieve very low complexity. Nevertheless the method handles only one mismatch queries ($L_W(S,\tilde{S}) \le 1$) what is not applicable to our problem (length of the suffix could be more than 1). We suggest using the method presented by Cole and Lewenstein which uses so-called the longest common prefix data structure to organize the vocabulary. The method can handle cases with edit distance more than one and has query complexity

$$O\left(m + \frac{(c\log d)^k}{k!}\log\log N\right) \tag{21}$$

which is less than $O(N\log N)$ and $O(N\cdot m)$ for $k,m << n$, where d – is the number of strings in the dictionary, k – specified maximum (preferred) distance in the query, c – some constant. Since, legitimate prefix is expected to be small in an anomalous sequence, the preferred edit distance k would be also small. The only modification in the algorithm concerns the objective function, which must be changed to weighted edit distance depicted in expression (18).

After defining the closest group of anomalies, the system must analyze the pattern of propagation of anomalies in the group to reveal the replication feature. The replication property is determined through the processes connectivity pattern. The connectivity pattern (or connectivity graph) is represented by weighted directed graph $C(V,E)$ with nodes being processes in different hosts and edges presenting last interaction. If one process sends some data to the port of another process it listens to, we assume that the sender process interacted with the second one. Thus, if a process *one* interacted with the process *two*, the system adds edge to the graph (or upgrades it if there is already one) with the weight equal to the relative interaction time.

This algorithm provides score which is compared to threshold to decide if anomaly propagation is indeed replication. The score takes into account the relative number of instances matched to replication pattern and shows how much the propagation similar to replication pattern. If the size of the group exceeded some size threshold and score is still pretty low system will declare false positive. Since only false positive may have many instances (unknown operation phase massively turned in many hosts) and not have propagation pattern.

4.2. IDS Implementation and Practical Improvements

Based on the models presented in the previous sections, the authors have implemented an IDS operating in MS Windows environment. The IDS consists of a client part and a server part. The client agents (installed on every host covered by the IDS) monitor system calls invoked by selected processes to detect anomalies. A server application receives abnormal sequences from the clients and performs anomaly propagation analysis.

During IDS development, we introduced several practical improvements to the original version. In order to decrease the overhead, the IDS client does not monitor and analyze system calls of the selected process until the process interacts with a remote host. After detecting such an interaction, the IDS client analyzes system calls to establish the fact of normal process operation or abnormal activity. Then the system returns to the waiting mode. Currently, we use the WinPCAP library to detect 3-way handshake of TCP session. The authors need to point out that the system only analyzes SYN packets captured and reported by the WinPCAP driver working in kernel-level packet filtering mode so the overhead for the user-level packet processing is minimal.

During the attack the system calls of the shell code would be executed in the thread which invoked the susceptible function being exploited. Therefore, all shell code system calls would have one thread ID. Since proposed anomaly detectors use process specific model, “malicious” system calls may be interchanged with possibly different normal calls invoked by other threads from generation to generation of the worm. Therefore, the server separates received call sequence into thread-specific subsequences before analyzing.

The authors do realize that an anomaly propagation pattern could be specifically factorized if the worm were to cyclically use a different shell code at successive generation steps. To address this problem we propose to extend propagation analysis through combining graphs and analyzing the obtained general graph. This graph will have nodes belonging to one of the combined groups. Thus, if the worm propagated using different shell codes, the combined graph will preserve propagation structure demonstrating an alarm propagation, which may reflect several different anomalies.

The authors are also aware of methods to obfuscate propagation detection through random insertion of dummy system calls in the propagation engine, and will address this issue in the future research.

4.3. Experimental Studies

Experimental studies were conducted in two ways. First, an the stage of approach evaluation we performed simulation of the worm attack in MATLAB using system call traces of real worm and legitimate software. Such simulation allowed us to estimate the potential of the proposed system to reveal shell code traces within the legitimate activity of the software which is free of particular vulnerabilities and cannot be yet compromised by real worms in real environment. Moreover, it allowed us to estimate the threshold distribution of the worm detection in different simulated attacks.

Then we experimented with real worms in virtual infrastructure to assess the performance of our prototype implementation of the IDS utilizing Markov chains and server-level system performing alarm correlation.

4.3.1. Trace Based Simulation

The trace based simulations were conducted using real data comprising traces of system calls recorded from legitimate processes as well as malicious software. For the malware we chose the forth generation of the Sasser worm – Sasser.D worm. Several legitimate processes including Microsoft Internet Explorer were also used.

We gathered data from the processes in various operation modes and crated traces of system calls. While the process monitoring could not be claimed to be comprehensive, the size of records comprised tens of thousands calls. For instance, we utilized 50,000 system calls issued by Internet Explorer to compute Markov models. We also recorded several contiguous system calls invoked by a victim process executing the shell code of the Sasser worm. Hence, this segment of system calls constituted the worm trace.

After obtaining Markov models, we performed discrete time simulation in MATLAB. In the beginning of the simulation, 500 fictitious homogenous processes, running on 500 imaginary hosts, were assigned random starting index of system call in the trace (pool) of the recorded calls from some fixed legitimate process. Then we assumed that processes start execution form the assigned position. The rationale behind such simulation is that for a sufficiently large trace, it is reasonable to expect that any starting point distribution may happen in real life. In other words, if the trace of a process is really long, then this process in any host will eventually pass through some segment in the trace during the corresponding operation phase in real life.

Simulated processes running in 500 hosts invoked system calls from the general trace one after one starting from the assigned position. The delay period between two subsequent invoked calls was generated at random. This helped to reflect different delays of system calls execution due to the operation system overload that frequently happens in real life, especially during massive viral attack. To simulate the connectivity between processes, every process's connection to another process was chosen at randomly with some stochastically changing periodicity.

An attack was simulated through inserting Sasser's segment in front of the current trace position of target process. The target process virtually invoked the worm trace and continues invoking system calls from the legitimate trace, what reflects the normal execution return in real life. The attack pattern was specified in advance and performed with corresponding virtual inter-process connections.

The Sasser.D worm is indented to attack LSASS process, but today this vulnerability is well known and any antivirus software can detect it. Since system calls invoked in the penetrated process reflects not the exploit itself, but the shell code, we assumed that the same shell code may be used in other attacks for different process using different exploits. Thus, we decided to simulate attack on the Internet Explorer. On the other hand, the Internet Explorer was chosen for simulation because it is a quite complicated process with respect to modeling in system call domain.

Simulated attacks were in slow warm fashion. 30 different attacks were emulated, and all of them were detected at early stages.

The results of one of the attack detection are presented below. The propagation score of one of the attacks is depicted in the Figure 14. It shows 12 groups arranged in the score descending order. The first group contains indexes of attacked hosts. One can see that the other "normal" groups have score five times lower than score of the attacked group. The attack was detected after four infected processes, what shows agility of the detection scheme in spate of the lots of noise factor provided by connectivity simulation.

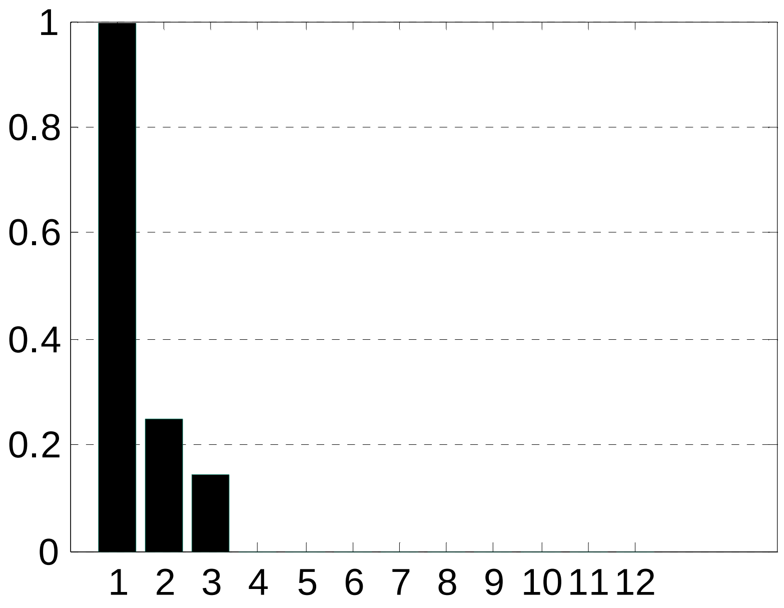


Figure 14. Propagation Score

Figure 15 shows the local false positive rate of those four processes which participate in the attack pattern. This false positives illustrate the host-based detection without propagation analysis, decision made on hosts before sending anomaly to server. One can see that for threshold less then 20 all of the hosts have high local false positive. We repeated in offline the same attack but with local thresholds ranging from 10 to 20. In all cases attack was successfully detected and no global false positive observed, what shows robustness of the network level detection.

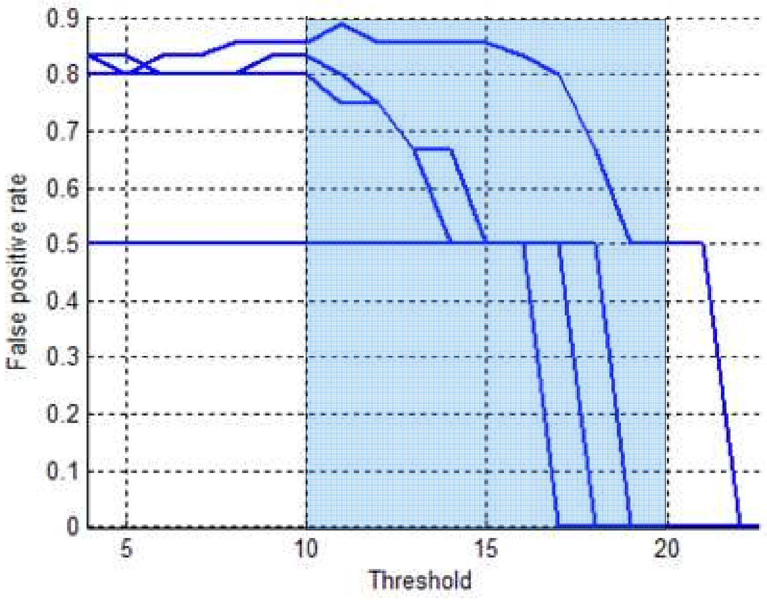


Figure 15. False Positive Rate for Different Thresholds

Figure 16 illustrates the predicting performance of one of the attacked hosts. The big wide spike is due to Sasser trace. In this case a worm anomaly has a high score, but likelihood ratio depends on model quality what may end up in good prediction for worm trace (in case of bad model) and as a result in false negative of host based detection. One can see that for threshold 15 stride dashed line, even if the statistic would be truncated down to shaded region (small dashed) what is three times less, the worm trace would anyway be detected locally and sent to the server. And as it was mentioned above attack for local threshold from 10 to 20 (including 15) was successfully detected. This simulated result shows that even for bad model the network level detection can still reveal the attack.

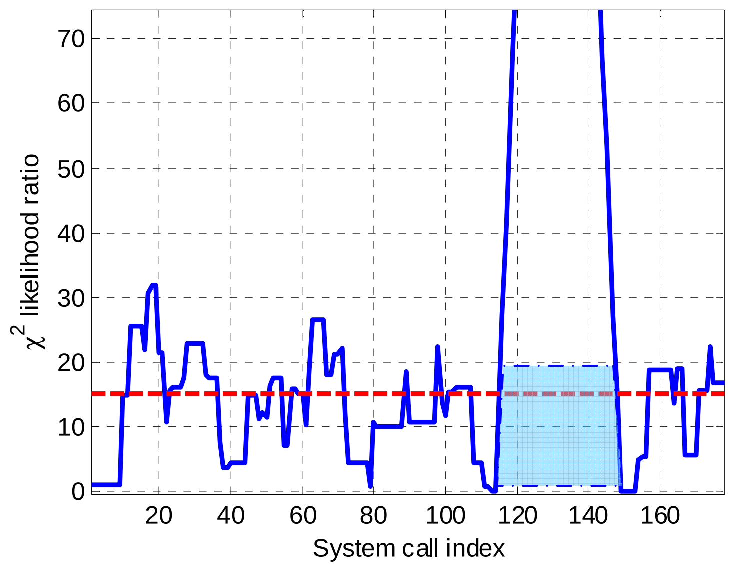


Figure 16. Anomaly Score (Predicting Performance)

4.3.2. Real Environment Experiments

“Real life” experiments were performed utilizing the Binghamton University network testbed [15] hosted 200 virtual machines with vulnerable versions of Windows XP. We experimented with two real worms W32.Welchia.G and W32.Shelp.A.

According to Symantec.com W32.Welchia worm is notable for its high geographical distribution and traffic load. For our experiment, we had to reverse-engineer and modify the original worm. In order to make its propagation faster we removed ISS exploit, date checking, and Blaster worm subverting engine. Additionally, we restricted the victim scan space. The modified version exploits the RPC DCOM vulnerability and scans only vulnerable and reachable hosts resulting in an increased propagation rate.

W32.Shelp.A worm was reverse-engineered deeply up to the level of extracting shell code and exploit buffers. The shell code was incorporated into educational worm. As a result we obtained the controllable worm with original propagation engine possessed by W32.Shelp.A worm.

The predicting capability of non-stationary models versus stationary models for several legitimate processes (services) including: Svchost (RPC DCOM), Internet Explorer, LSASS, CCAPP was presented in the previous Section. Then we launched W32.Welchia worm 20 times with an arbitrarily chosen attack deployment node. We set the threshold for replication score to 0.75 and the minimum size of the group to 5 hosts. In Figure 17, the solid line (left axis) shows the number of infected hosts before anomaly propagation detection and the dashed line (right axis) determines the mean distance between the string representing the group that raised an alarm and the members of the group. During every attack instance the system detects anomaly propagation after 5 to 13 hosts have already been infected and on average after 8 hosts have been infected. The mean distance fluctuates from 0.05 to 0.18, which demonstrates a high member similarity. We did not observe any false positives (group spawned by non-worm sequences being attributed to propagation), which demonstrates the high performance of the IDS.

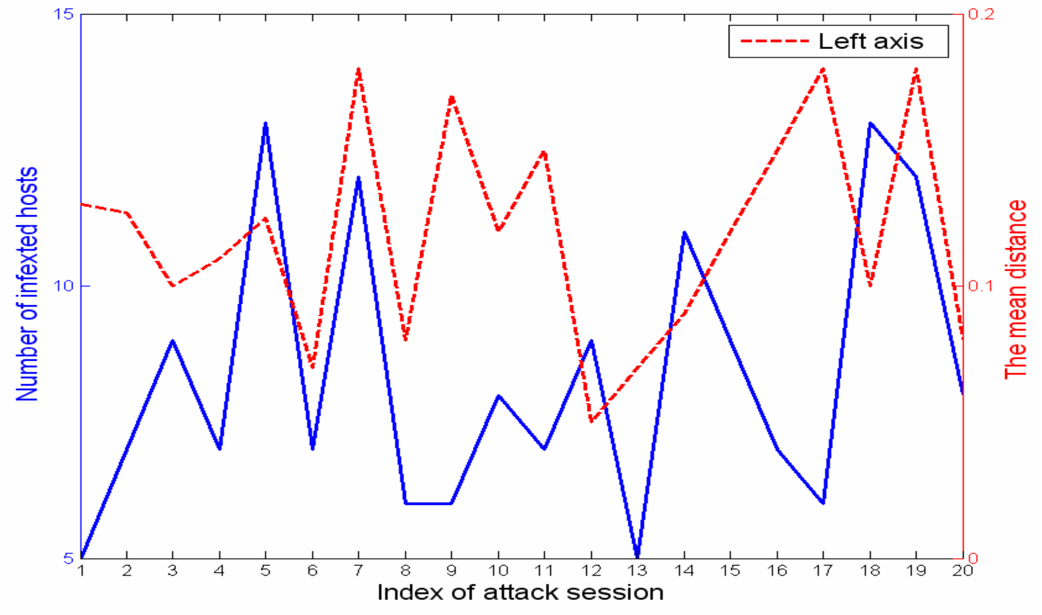


Figure 17. Number of Infected Hosts and Anomaly Group Score upon the Detection

W32.Shelp.A worm employs executable download and execute propagation engine and has very intense activity in terms of system calls. Its shell code invokes high level API functions such as “InternetOpenURL” and “InternetReadFile” to establish connection with remote host. Then the shell code downloads worm image through http protocol what results in long and highly repeatable chain of system calls. Our system demonstrated even higher performance with such propagation engine preserving mean group distance around 0.1 and number of infected hosts around 6 hosts.

For the arbitrarily chosen infected host #14, Figure 18 shows the likelihood function (LF) recorded during the attack. One can see that the attack is detected reliably (wide spike) and the segment having a LF that is higher than 100 was treated as an anomaly and reported to the server.

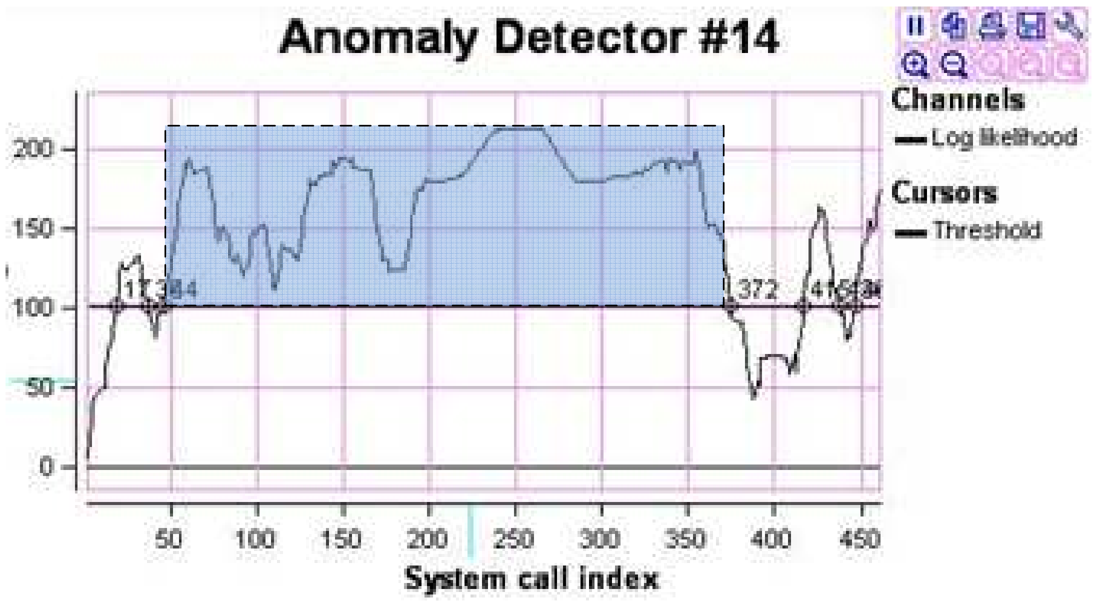


Figure 18. Likelihood Function

4.4. Server-Based IDS

The above results justify the development of a network oriented intrusion detection system (IDS) based on monitoring hosts behavior in terms of inter-host communication and activity. The system distinguishes suspicious activity of a process currently running on the given host by analyzing system calls invoked by the process. The IDS is designed to monitor the *network* system calls as well as to analyze their input parameters and flags what allows the IDS to reveal suspicious system calls quite dependably. The advantage of the IDS is that it is able to recognize inter-host suspicious activity inheritance in terms of behavior commonality among suspicious hosts using “malicious” system calls. The system could consist of two levels, the host level and the server level. The host level is represented by the hosts equipped with a special software module which monitors and pre-selects suspicious system calls and reports them to the server level. The server level software maps suspicious system calls on pre-defined attack graphs, computes the distance between suspicious sequences and corresponding attack graph, constructs so-called suspicious activity tree and derives hypothetical joint probability of false positive thus providing a score to the intrusion detector.

We present two approaches on which IDS could be based: the inter-host based and network based. Inter-host detection is based on system calls monitoring and recognizing potentially “malicious” network system calls provided with certain input parameters. System calls are monitored for every process being active in the host system and detection decisions are made using the local information regarding the host. We should point out that this approach is not an anomaly detection which is usually prone to have high rate of false positives due to system state changes (installation of a new application, previously unseen legitimate behavior of the processes). We rather look for special behavior types stipulated by certain system calls pre-defined by an expert. However, the system detects vulnerability exploiting the phase perpetrated by malicious software, so it is not a signature-based approach in its classical meaning.

Regardless its realization, a virus invokes certain system calls to exploit a vulnerability. For instance, replication (attack) procedure of buffer overflow regardless virus implementation details would anyway use appropriate network system calls to establish connection to victim host and send malicious code. An IDS expert is not required to know viruses or worms a priory; but instead he/she must just encode currently existing exploits in terms of system calls. Then the IDS would be able to detect new viruses provided they use known type of exploit. Since virus makers rarely invent conceptually new types of exploits, new viruses would have common "system calls signature" that is crucial for the success of the IDS. Such system calls invoked by the virus or worm would have special ("malicious") input parameters what is used as one of the indicators of suspiciousness of the given process.

The network-based approach takes into account comprehensive information collected from the set of hosts serviced by IDS. Knowing suspicious scenarios derived from different hosts (processes) and time stamps of system calls associated with appropriate attacks, we can recognize similar activity pattern in terms of achieved goals. Even if the virus is metamorphic (i.e. every generation has a different implementation) it will achieve the same goals during its propagation. These purposes may be recognized due to post-preconditions analysis. Having suspicious activity pattern one can define the score for detection procedure.

4.4.1. Inter-Host Based Detection

Consider a virus replication procedure in Win32 OS that involves some certain API functions (system calls) to be invoked. Replication of a network virus from one host (attacker) to another (victim) utilizes special kind of API functions – network system calls which are intended to do affect in some ways the victim process. We can recognize the direction of such system calls in terms of source (invoking process) and destination (targeted process). System calls themselves considered to be suspicious if some certain attributes (flags) are used as input parameters. In order to copy itself, the infected process invokes certain system calls, directed to the victim, with "malicious" parameters in definite sequence. One can name such a sequence an attack sequence. We distinguish preconditions and post-conditions of stages of the infection process. Each "malicious" system call in the attack sequence requires particular preconditions (provided by successful implementation of previous function in the attack sequence) and creates post-conditions. Sequence of pre and post-conditions along with attack sequence forms an *attack scenario*. Monitoring system calls may allow us to recognize such attack sequences. By analyzing system calls directed from one process (host) to another we can realize that the same malicious system calls (with suspicious parameters) occur in the sequence consistent with attack sequence and results of the system calls correspond to appropriate post-conditions in attack scenario, consequently we can claim that infection takes place.

Different attack scenarios can be composed a priori and represented by *attack graphs* where edges are system calls (with certain input parameters) and vertices are post-pre-conditions. An example of an attack scenario in its general form is showed in Figure 19. It shows five intermediate conditions and starting condition (victim is accessible) and final condition (victim has been infected).

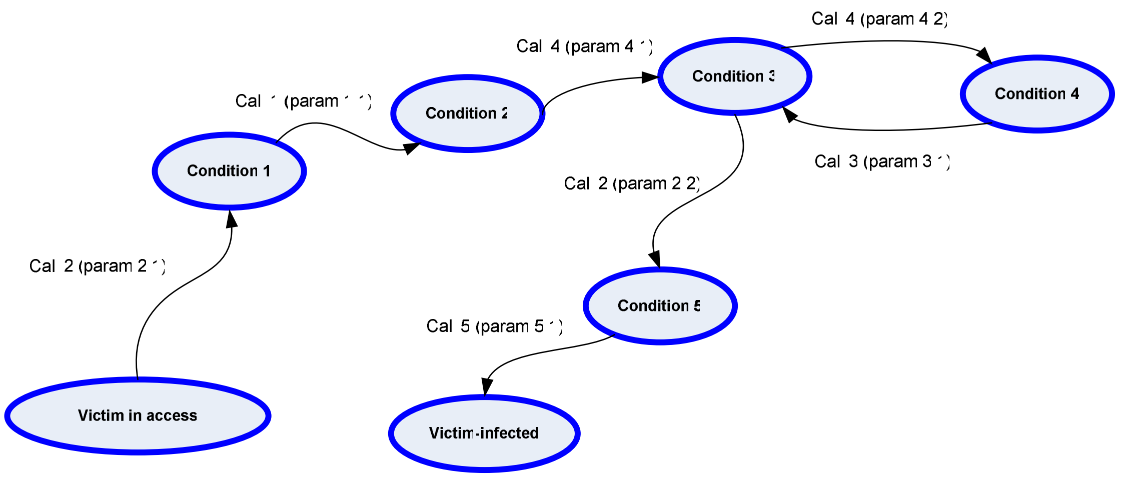


Figure 19. Attack Graph

Knowing certain attack graphs one can detect a suspicious activity on the graphs. A system call in the given sequence would be mapped on an appropriate edge of an attack graph if its pre-post conditions are consistent with the endpoints (vertices) of the corresponding edge. Mapped system calls sequences on the attack graph represents current suspicious scenario. We should point out that each formed suspicious scenario would have direction in terms of source (attacker) and destination (target).

The attack graph most similar to current suspicious sequence may represent current attack scenario. Knowing attack scenario (graph) one can even predict the consequent activity of malicious (infected) host and block certain system calls (relevant to the attack scenario) to be invoked by adversary host.

To measure similarity between an attack graph and current (observed) suspicious graph (scenario) we must introduce the concept of distance. Before deriving the formula for distance we should establish a measure of similarity of the graphs, i.e. a metric of similarity. The more two graphs G_1, G_2 are similar the greater the metric value has to be. Having metric in analytical form $M(G_1, G_2)$, one can define the distance in the following form:

$$d(G_1,G_2) = \left(M(G_1,G_1) - M(G_1,G_2)\right) / M(G_1,G_1), \qquad (22)$$

where $M(G_1,G_1)$ is the maximum possible value of the metric for graph G_1.[2] The metric must be indicative to special features of attack scenario similarity. After embedding suspicious

[2] Similarity metric of graph G_1 with itself

sequence (graph) into the attack graph, we take into the account following indicators to compute the metric:

- Quantity of common (consistent) vertices and edges of attack graph and suspicious graph
- Layout position and importance of the consistent vertices and edges (system calls) in the attack graph
- Connectivity of the common vertices in the attack graph

The *first indicator* shows closeness of the graphs strictly in terms of shared vertices and edges. The distance based on this indicator is analogous to binary Hamming distance.[3] However, the metric based only on the first term does not evaluate the importance of the consistent (shared) vertices or edges. In other words, some system calls may have key value in the attack scenario, while others may not be so important and may be substituted by different system calls. Therefore, we assign weights to individual edges in the attack graphs according to their positions and importance what provides the *second indicator* for the distance. For instance in Figure 20, there are two different suspicious sequences (black solid line) embedded on the same attack graph (blue dashed line). Let's assume that system call 1 is vital for virus replication (for example OpenFileA, CreateFileA or WriteFile) then this call is assigned high weight. The sum of weights of the edges *included* in suspicious sequences would be greater for the left one, because it includes the "heaviest" edges. This sum would constitute in some sense the metric between the attack graph and the given sequence. Hence, the left embedded scenario is more similar to the attack graph than the right one with respect to system calls importance.

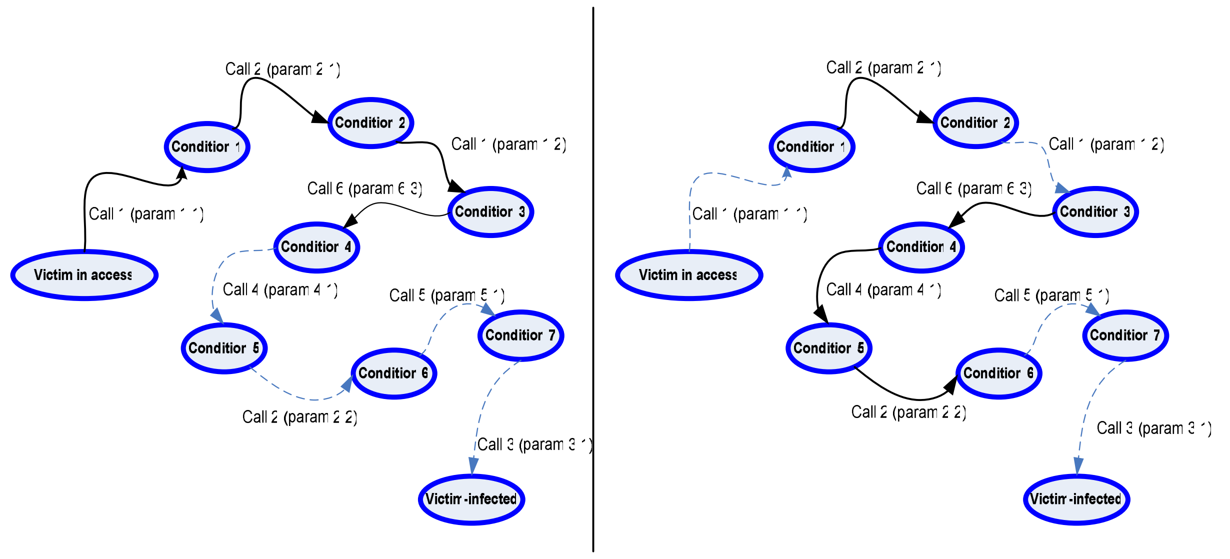


Figure 20. Two Embedded Suspicious Sequences on the Same Attack Graph

[3] Humming distance shows the number of different elements in two compared strings having equal length

The *third indicator* used to compute the distance reflects the cohesion of system calls of the suspicious sequence. While system calls and their importance are taken into account, it is also reasonable to distinguish lengths of sequences. In other words, suspicious system calls must be strictly connected in terms of post-pre conditions. Thus, after mapping the given sequence on an attack graph, we consider the continuity of the common part. Consider Figure 20 again. The total number of edges in two cases is the same - four. System calls of the left sequence are embedded in one graph

$$G_l(V_l, E_l)\ \ V_l = \{Victim\ in\ access, Condition\ \ 1,2,3,4\}, \quad (23)$$

and system calls of the right sequence are distributed among two sub-graphs,

$$G_{R1}(V_{R1}, E_{R1}),\ V_{R1} = \{Condition\ \ 1,\ 2\} \quad (24)$$

and

$$G_{R2}(V_{R2}, E_{R2}),\ V_{R2} = \{Condition\ \ 4,5,6\} \quad (25)$$

Thus, the second sequence has a gap in the scenario (Call 1 (param 1.2)) what assures the success of such an attack and makes it less suspicious. In order to make the metric indicative of the third term, we can increment the weight of every system call of the given sequence to a value proportional to the number of edges of a sub-graph in which the system call is contained.

Concluding the considerations stated above, one can derive analytical form of the metric of similarity between an attack graph $G_A(V_A, E_A)$ and a given (suspicious) scenario $G_S(V_S, E_S)$.[4]

$$M(G_A, G_S) = \sum_{e \in Es} \left(w(e) + k \cdot L(e)\right), \quad (26)$$

where,

$$\text{for } \forall e \in E_s \Rightarrow \begin{cases} w(e) - weight\ of\ the\ edge \\ L(e) = \left|G_i^{sub}\right|_E,\ e \subset G_i^{sub} \subset G_S \end{cases} {}^{5} \quad (27)$$

k - sensitivity coefficient for the third indicator (cohesion degree)

$L(e)$ - number of edges in the sub-graph in which edge *e* is contained

[4] $G_S(V_S, E_S) \subset G_A(V_A, E_A)$ - sub-graph representing scenario embedded on the attack graph

[5] G_i^{sub} - *i*-th connected sub-graph of the sub-graph $G_S(V_S, E_S)$, $|G|_E$ - number of edges in the graph *G*

By adjusting coefficient *k* one can obtain the optimal indicative power of the metric with respect to the continuity of the suspicious scenario. Weights of the edges in the attack scenarios must be assigned by an expert. Apparently, such network system calls as NtOpenFile, NtCreateFile, NtReadFile and especially NtWriteFile must be assigned heavy weight.

We should point out that distance is determined by the following formula:

$$d(G_1, G_2) = 1 - \frac{M(G_1, G_2)}{M(G_1, G_1)} \qquad (28)$$

By comparing the distance between current suspicious scenarios and different attack graphs with some specified threshold, one can detect the infected process at its replication stage. If several attack graphs are close to the sequence in question so that the distance is less than the threshold, we take the closest one and derive a hypothetical probability of the suspicious host being infected. The probability can be computed by the formula:

$$P_{\inf} = \frac{\tau - d}{\tau}, \qquad (29)$$

where τ - threshold. By choosing the value of threshold, we can adjust the level of desired awareness.

4.4.2. Network Based Detection

As it has been stated above, by adjusting the distance threshold, one can achieve the optimal level of suspiciousness. However, a high threshold may result in the increased number of false positives. In order to decrease false positives we must consider not only pair-wise connections of server processes, but also the network of such processes in terms of suspicious activity inheritance. In other words, if we suspect a host to be infected, the destination of the activity of the suspected host must be monitored to detect similar abnormal behavior in terms of achieved goals. By the goal of suspicious scenario we understand the obtained post-conditions during the scenario and of course the final condition of the whole scenario. If the second host/process tends to invoke a suspicious scenario of system calls (which generally speaking can be different) resulting in the same outcomes (conditions), we can claim that this host inherited the next generation of the worm virus. Consider for example, slow worm propagation topology presented in Figure 21. Let's assume that the replication (attack) scenario of a worm does not completely match any known attack graph, but is close enough to one graph so that the distance is less than the threshold τ and the hypothetical probability of infection $P_{\inf}$=0.6. Then hypothetical probability of false alarm is 0.4, i.e. ($P_{false} = 1 - P_{\inf}$). Apparently, Figure 21 shows three generations of suspicious activity. Assume that host 1 invoked a suspicious sequence which first was directed to hosts 2 and 3 (so the probability of false alarm is for each direction is 0.4). Then after some time, hosts 2 and 3 start invoking the same sequence. This would be the second generation of suspicious activity. After that, host 5 exhibits the same behavior, which constitutes the third generation, so the probability of false positive after the third generation would be:

$$P_{false} = P(F_{12}, F_{25}, F_{56}) = P(F_{12})P(F_{25})P(F_{56}) = 0.4^3 = 0.064 .^{6} \qquad (30)$$

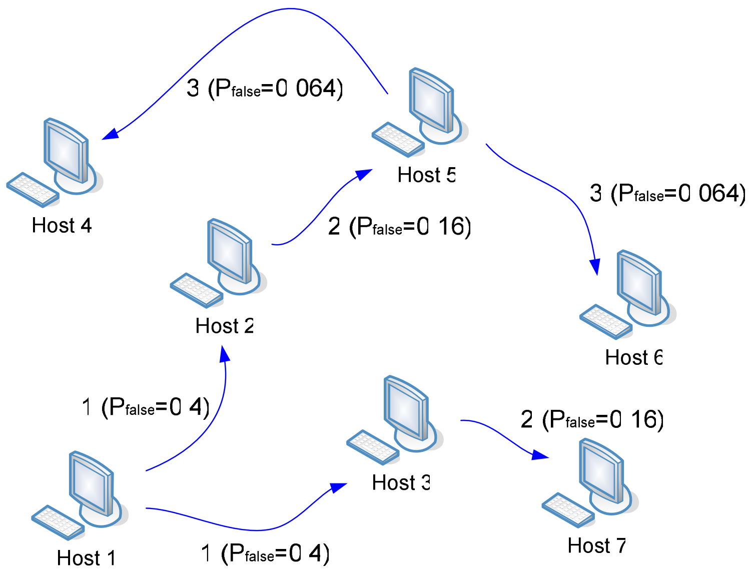


Figure 21. Slow Worm Generations

We should point out that if re-infection is not possible, slow worms would have a tree-like propagation topology. Therefore, suspicious activity generated by these worms would also have a tree topology. Such a graph (tree) provided by weights equal to corresponding false positive probability will be called a *suspicious activity graph (tree).* Thus, knowing the number of generations one can define the hypothetical probability of false alarm:

$$P_{false} = \min\{\prod_{x \in L} P(x) \mid L \in \mathbf{L}\}, \qquad (31)$$

where, $\mathbf{L}$ - is the set of various directed paths belonged to suspicious activity trees , and *P(x)* – false alarm probability of suspicious sequence represented by edge *x*

The latter formula can be implemented using the Breadth First Search algorithm [21] on the suspicious activity graph. The only modification required by this algorithm is a labeling technique. Each vertex should be labeled by the value which is equal to production of ancestor label and weight of its edge.

[6] $P(F_{xy})$ - “local” probability that the host *x* infected the host *y*

In summary, the so-called inter-host intrusion detection is based on revealing suspicious system calls relevant to virus replication and determining the distance to certain attack graphs. The value of the distance allows us to recognize infected process. However, such method may lead to false positives providing the suspicious scenario does not match any known attack graphs. In order to avoid false positives, the network based detection is proposed. Such detection is based on monitoring the inherited activity. Then we compose so-called suspicious activity tree and calculate joint probability of false alarm that seems to be pretty dependable.

4.5. Intrusion Detection System Architecture

The architecture of the Intrusion detection system (IDS) is depicted in Figure 22. The system has a server part and a host level. The server part could be implemented on a server machine. The IDS software installed on the server would solve the most computationally-intensive tasks. The host level software would be cloned on every user machine covered by IDS and would collect and forward some host data to the server.

The host level is represented by process data monitor subsystem. Every host served by the IDS must have this subsystem activated. The subsystem collects suspicious system calls with parameters invoked by processes as well as some achieved system conditions and forwards them to the IDS server. The subsystem consists of three modules: process data monitor, conditions reader and system call selector. The process data monitor module listens to system calls invoked by the processes and selects certain call types. It merely checks out if the system call belongs to the set of interest and passes them to other two modules of the subsystem.

The conditions reader module analyses the result (output parameters) of some system calls provided by the process data monitor and recognizes special conditions used in pre-defined attack scenarios.[7] A condition is considered to be achieved, if certain system call with particular parameters has been successfully invoked. As soon as the module determines special condition of interest it sends it to the IDS server tagged with the time stamp.

The system call selector module is responsible for selecting potentially suspicious system calls. Pre-selected system calls, obtained from process data monitor, are checked on having “malicious” input parameters. If certain system call has suspicious parameters, the module sends appropriate data signal to the IDS server. The data signal reflects the system call itself, its input-output parameters, and the time stamp.

Since each machine serviced by the IDS would have host level modules to be installed, the module distribution between levels is such that IDS software on the host level would consume little amount of computational resources. The process data monitor module makes only comparison operation thus consumes little resources. Since verification of input parameters

[7] Attack scenarios are described in the previous section.

implies comparing them against pre-defined sets of suspicious input attributes, the system call selector and condition reader modules could be quite computationally efficient.

The server level is represented by four modules each implemented on the IDS server. Before evolving the IDS, attack graphs must be composed by an expert off-line. The graphs are stored in the IDS server in a special succinct format thus allowing the system to use them directly without decompressing. The system call superposition module maps current suspicious system calls and system conditions on all appropriate attack graphs and suspicious scenarios. As input data, this module uses system call signals and current system conditions reported by process data monitor from each host in the IDS network. Based on this data, the module generates new and updated suspicious scenarios (mapped system calls along with system conditions) which are saved in the database. For example, when the module obtains a system condition from host *i* it looks trough the suspicious scenario database and superposes the condition on every consistent condition in the scenarios if the condition appears later than the last condition in the appropriate scenario. The module also looks through attack graphs and maps the condition on appropriate attack graph forming new suspicious scenarios.

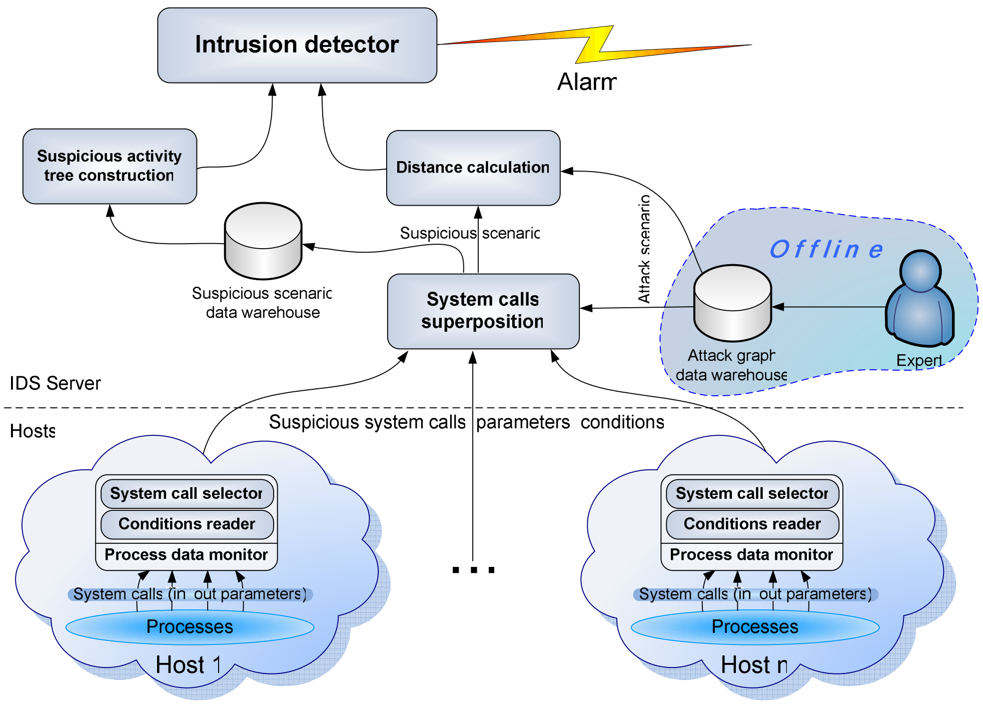

Intrusion detector
Alarm
Suspicious activity tree construction
Distance calculation
Suspicious scenario
Attack scenario
Offline
Suspicious scenario data warehouse
System calls superposition
Attack graph data warehouse
Expert
IDS Server
Hosts
Suspicious system calls, parameters, conditions
System call selector
Conditions reader
Process data monitor
System calls (in, out parameters)
Processes
Host 1
. . .
Host n

Figure 22. IDS Architecture

Suspicious scenarios are stored during some period of time and after this time elapses they are to be erased. Such suspicious scenarios are formed for every monitored process invoking suspicious system calls in any host serviced by the IDS. Suspicious scenarios accompanied by the appropriate time stamps and source/destination descriptors are forwarded to the distance calculation module and suspicious activity tree construction module.

Distance calculation module is responsible for computing the distance between mapped suspicious scenario and appropriate attack graph; it also defines a hypothetical probability of corresponding process to be infected. Using scenario invoking time stamp and the direction descriptor, the suspicious activity tree constructor recognizes topologies of suspicious activity inheritance and constructs so-called suspicious activity trees. These trees are aggregated up to inter-hosed level and transmitted to intrusion detector. Intrusion detector assigns weights to every edge of the suspicious activity graph according to the hypothetical probability of process being infected. Then it calculates the joint probability of hypothetical false alarm of virus propagation. The probability of false alarm is reported to the system supervisor and compared against assigned threshold, if its value less than the threshold, an intrusion alert is generated.

4.6. Simulation of Connectivity within a Network

The utilization of the experimental network testbed for the implementation and validation of the developed software systems requires random e-mail-type network generation. The algorithm below provides the specified degree of connectivity distribution and desired clustering level. Only three parameters are used as input variables: the number of nodes (users), extra probability for double-path adjacent vertices, and extra probability for triple-path adjacent vertices.

4.6.1. Models of Topology of E-Mail Networks

E-mail traffic (network) can be represented by a directed, weighted multi-graph $G(V,E)$, where each vertex is represented by a user, and an edge (u,v) is added if the users u and v corresponded through e-mail over specified period of time. Such a graph is composed over certain period of time in the way that every edge (u,v) has a weight which is equal to the number of e-mails sent to *v* from *u* during the period. Thus, e-mail network can be simulated through random graph generators. Before creating a random graph we must study features of e-mail graph topology.

One of the most obvious ways to characterize the graph topology is to obtain a degree distribution. Degree distribution of a graph shows the number of nodes having the same degree with respect to every possible degree value of the graph. Formally, it may be represented in the following way:

$$D(x) = \sum_{u \in V} \delta_{\upsilon(u),x} \, , \tag{32}$$

where $\upsilon(u)$ - degree of vertex *u*, and

$$\delta_{x,y} = \begin{cases} 1, x = y \\ 0, x \neq y \end{cases} \tag{33}$$

The topology of e-mail network graphs has been studied in [22], [23]. According to the papers the degree distribution of e-mail traffic graphs follows Lavette's law:

$$D(x) = c \cdot \left(\frac{N \cdot x}{N - x + 1} \right)^{-b} \, , \tag{34}$$

where $D(n)$ the number of nodes of n-th degree,

N – maximum degree,

c,b - fitting parameters.

Based on the distribution we may generate random graph. There are two main approaches to generate graph randomly [24, 25]. The first one is a random graph generator which stars with unconnected set of vertices and performs consequent interconnection of arbitrary vertices. The second approach allows creating random graphs possessing *small world* properties. A small world type graph has high probability that two neighboring vertices are incident. Thus, such graph has big cluster coefficient:

$$C = \frac{1}{|V|}\sum_{u \in |V|} C_u \, , \; C_u = \frac{2|E_u|}{k_u \cdot (k_u - 1)}, \forall u \in V \, , \tag{35}$$

where C_u - cluster coefficient of vertex u , E_u - number of edges of the sub-graph induced (composed) by k_u incident vertices to the vertex u . Small world graph can be constructed using Watts-Strogatz's [25] model or Kleinerg's model. Watts-Strogats model starts from the ring with consequently connected nodes and then adds new edges connecting two randomly chosen vertices.

The mentioned methods are relatively simple to implement, nevertheless they do not provide specified distribution. A method for generating random graph with assigned distribution is proposed in [24].

The following algorithm based on [24] allows us to create a random graph $G(V,E)$ with degree distribution $D(x)$:

1. Compose a list L containing every vertex u $\upsilon(u)$ times. Thus $|L| = \sum_{u \in V} \upsilon(u)$
2. Permutate the list L and split it into two equally sized sub-lists L_1, L_2
3. Assign pairwise incident relation between components of the lists $L_1(i) \rightarrow L_2(i), \; i = 1..|L_1|$

The composed graph may have loops what does not take place in e-mail network graphs. Moreover, such graph may have more than two edges possessing the same endpoints.

4.6.2. E-mail Network Generator

Algorithms described above do not meet all requirements of an e-mail network. In order to meet these requirements, the graph must be constructed according to connection probabilities of the nodes. The connection probability from node u to vertex v reflects the probability that the user u sends e-mail to the user v and it depends on in-degree of v. Thus, if user u sends an e-mail to any user from the particular set of uses S then the user $v \in S$ which has the maximal in-degree, will have the highest probability to be a recipient. Using these ideas one can derive more sophisticated algorithm for composing random graph which does not have parallel edges and has specified degree distribution:

$$L = \{(u, \nu(u)) \mid u \in V\}^{8}; \qquad (36)$$

Compose a set of tuples of vertices and their degree values

```
$A = \varnothing$

*while* $L \neq \varnothing$
    • Take any element $\tilde{m} = (\tilde{u}, \nu(\tilde{u})) \in L$
    • $L := L / \tilde{m}$ ; *Subtract* $\tilde{m}$ *from L*
    • Using $\left\{ \frac{l(2)}{\sum_{l \in L} l(2)} \mid l \in L \right\}^{9}$ as probabilistic model of the source L, chose randomly $\nu(\tilde{u})$
      elements from L and unite them into the set $L_S$
    • $A := A \bigcup (\tilde{u}, L_S(1))$ ; Form adjacent string for vertex $\tilde{u}$
    • $\hat{L}_S := \{(\hat{u}, \nu(\hat{u}) - 1) \mid (\nu(\hat{u}) > 2)(\hat{u}, \nu(\hat{u})) \in L_S\}$
    • $L = (L / L_S) \bigcup \hat{L}_S$ ; Get rid of vertices which used all their degree

*end*
```

This algorithms creates undirected graph, but the graph can be easily directed by choosing directions of each edge. Directions can be chosen randomly or may follow some specified distribution. For example, in e-mail networks the nodes with very high degree usually have the far larger number of out-edges with respect to the number of in-edges (university information services daily send thousands of e-mail to their clients). Thus, a graph composed using the latter method will have specified in-degree and out-degree distribution and will not have self-loops or parallel edges.

The last property of the graph to be taken into account is the clustering coefficient. According to [22], an e-mail network exhibits not only power law degree distribution but also small-world properties. Some models for small-world properties have been discussed above, but they do not support the Lavalette's degree distribution. Therefore, we must modify the random graph generator so that the graph would have desired clustering property. In order to do it, we must change the probability model defined for each iteration of the loop. Since small-world properties

8 $\nu(u)$ - degree of the vertex *u*

9 $if\ \ l = (u_l, \nu(u_l)),\ then\ \ l(2) = \nu(u_l)$ the second component (degree) of the element *l* of the set *L*

assume neighboring nodes to be incident with higher probability, we can redefine probabilistic model in the following way. Let *P* be a vector defining the probabilistic model used in the third step of the loop of the algorithm, formally

$$P = \left\{ \frac{l(2)}{\sum_{l \in L} l(2)} \mid l \in L \right\} \tag{37}$$

Then $$P_m = (P + P_2 + P_3) / \|P + P_2 + P_3\|_1 \ , \tag{38}$$

where P_m - modified probabilistic model (vector)

$$P_2(i) = \begin{cases} x \dfrac{P(i)}{\sum_k P(i_k)}, i \in D_2 \\ 0, \ i \notin D_2 \end{cases} \ , \tag{39}$$

- extra probability for second adjacent vertices

$$D_2 = \{ i \mid (\tilde{u} \xleftrightarrow{2} l_i(1))(l_i \in L) \}^{10} \tag{40}$$

- set of indexes of elements of *L* which vertices are connected to the given vertex $\tilde{u}$ through 2 edges (second adjacent vertices)

$$x = \frac{\sum P - \alpha_2 \sum_{j \in D_2} P(j)}{\alpha_2 - 1} \ , \tag{41}$$

α_2 - the input value that specifies the fraction of total probability which second adjacent vertices pull toward themselves, and

P_3 - extra probability for third adjacent vertices (can be defined analogically)

Thus, values α_2, α_3 determine the amount of additional probability which will be shared among the second and third adjacent vertices respectively. For instance inputs $\alpha_2 = 0.25$ makes the sum of probabilities of second adjacent vertices equals the sum to their original probabilities plus

10 $u \xleftrightarrow{2} v$ - means there is double path connecting *u* and *v*

$if \ l_i = (u_i, v(u_i)), then \ l_i(1) = u_i$ - first component (vertex) of i-th element of the set *L* additional 0.25. Adjusting coefficient α_2, α_3 we can achieve any desired graph cauterization coefficient. Actually high α_2, α_3 may result in disconnected graph having several isolated cliques or semi-cliques. Figures 23, 24 present connectivity matrix of a random graph and the graph itself created by the discussed procedure. The procedure was run for 400 vertices following Lavalette's distribution and $\alpha_2 = 0.2, \alpha_3 = 0$. In Figure 23 users (nodes) are indexed in degree ascending order.

In Figure 24 the heaviest users (high degree value) are focused in the center, and light users are shown on periphery.

One can see a very big cluster (bottom-right corner) and some different little clusters, what is natural for an e-mail network, so-called small-world property [22, 23]. In Figure 24, the big cluster is situated in the center. Moreover, there are some isolated pairs what is also matches small-world properties.

Actually, the big cluster is represented by the users having high degree value, usually they are relevant to mass media, news or information services within an organization. Small clusters reflect so-called communities of interests or communities of practice [26] which involve project teams, cheaters, and hierarchical (administrative) structures within particular department.

Figure 25 shows degree distributions of different random graphs and Lavalette's curve (theoretical distribution). One parameter of Lavalette's curve has been adjusted: $\beta = 1.1$. Thus the curve in our case has the following analytical form:

$$Y(x) = 76 \cdot \left(\frac{15 \cdot x}{15 - x + 1} \right)^{-\beta} \qquad (42)$$

The picture implies that the red graph ($\alpha_2 = 0.4$) matches theoretical distribution to a greater extent. Thus, it can be inferred that greater adjusting coefficient α ensures the network to be more realistic.

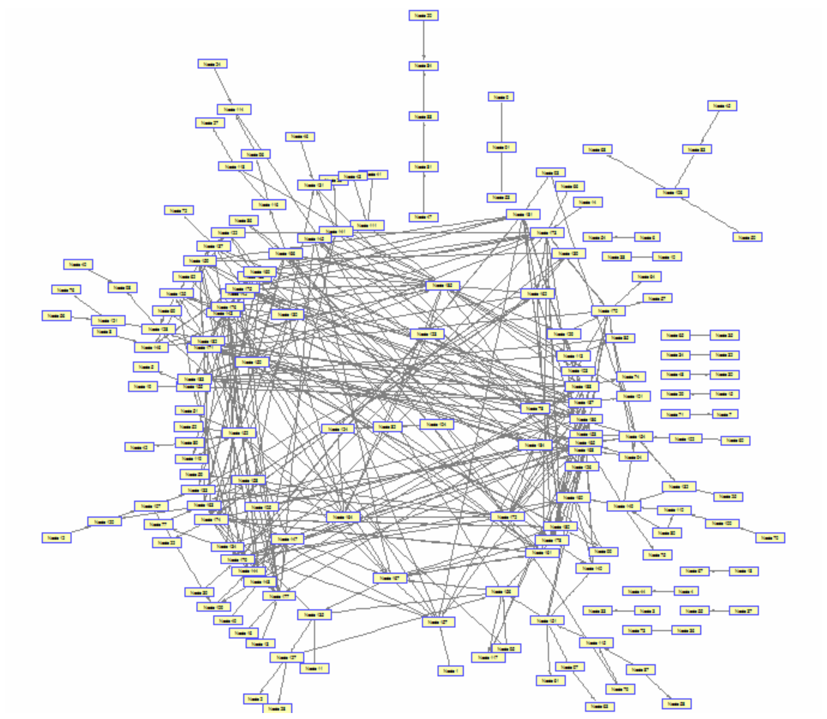


Figure 23. Random Graph (a=0.4, N=400)

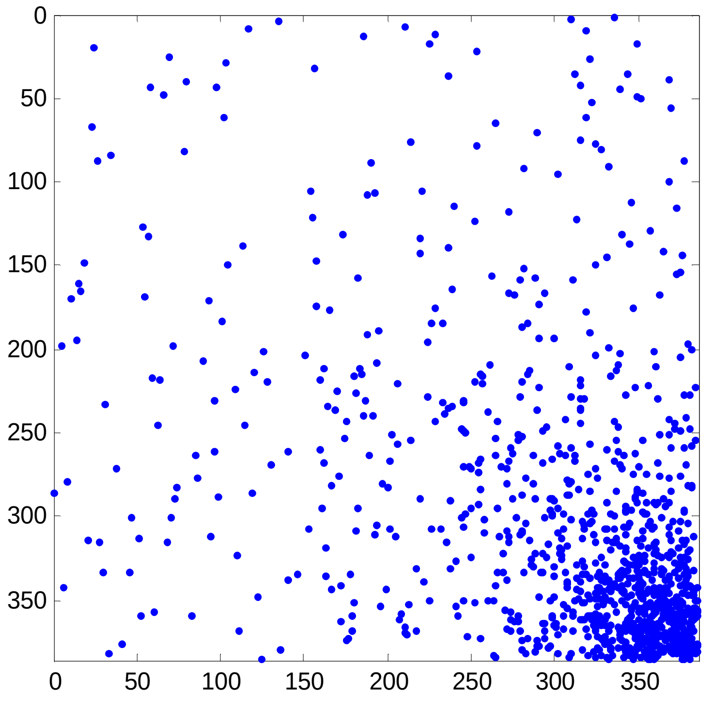


Figure 24. Connectivity Matrix (a=0.4, N=400)

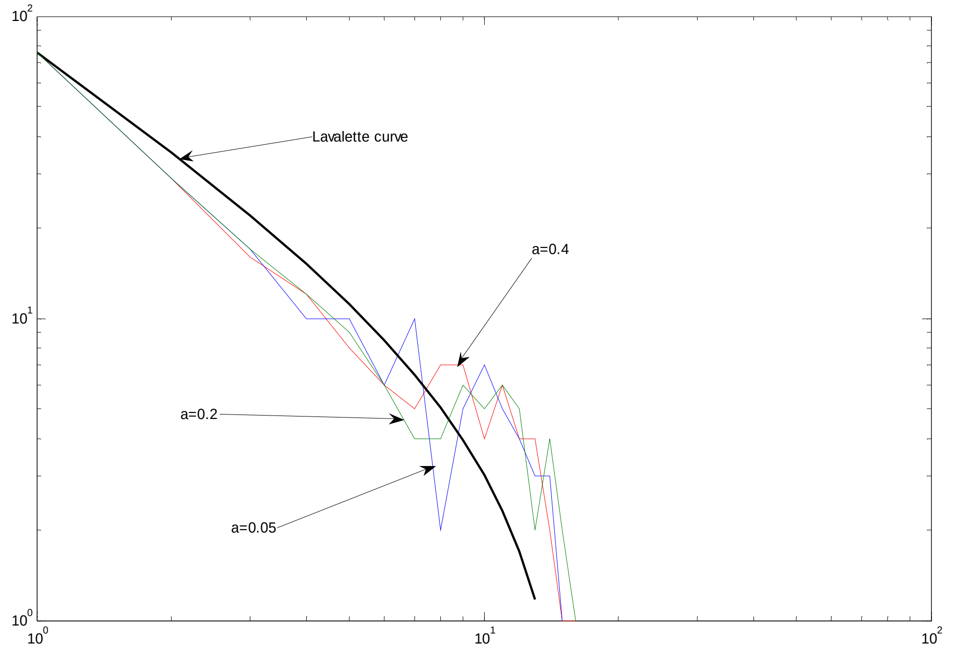


Figure 25. Degree Distributions

5.0 CONCLUSION

Our research demonstrated that the anomaly propagation concept, combined with the application of non-stationary Markov models, can provide a high level of confidence in attack detection. According to the experiments, we did not observe any false positives in the global network level sense. Also, anomaly propagation was detected at early stages of the worm attack, showing high dependability and low detection inertia of the IDS.

Future work will be focused on problems white list application and extensions for multipartite (many sources) attack detection. A multipartite attack must be treated in a different way and may not have tree-like propagation pattern. Propagation concept should be generalized to handle multi source patterns.

6.0 REFERENCES

[1] Michael Cobb. “Know your enemy: Why your Web site is at risk”, *URL: http://SearchSecurity.com*, Security School, Accessed Jan, 2005

[2] V. Skormin, A. Volynkin, D. Summerville, J. Moronski. “Run-Time Detection of Malicious Self-Replication in Binary Executables” *Journal of Computer Security*, vol. 15, no. 2, pp. 273-301, 2007.

[3] S. A. Hofmeyr, S. Forrest, and A. Somayaji. “Intrusion detection using sequences of system calls,” *Journal of Computer Security*, vol. 6, no. 3, pp. 151–180, 1998.

[4] A Durante, R Di Pietro, LV Mancini. “Formal Specification for Fast Automatic IDS Training” *Lecture Notes in Computer Science,* 2629:191-204, 2003.

[5] S. Stolfo, W. Lee, E. Eskin. “Modeling system calls for ID with Dynamic Window Sizes”, In *Proceedings of the DISCEX II*. June 2001.

[6] A. Liu, C. Martin. “A Comparison of System Call Feature Representations for Insider Threat Detection”. In *Proceedings of the 6th IEEE Information Assurance Workshop*, 2005.

[7] G. Tandon, P. Chan. “Learning Useful System Call Attributes for Anomaly Detection”, In *Proceedings of the FLAIRS Conference,* 2005.

[8] M. Xu, C. Chen, J. Ying. “Anomaly detection based on system call classification”. *Journal of Software*, 15(3): pp. 391-403, 2004.

[9] C. Kruegel, D. Mutz, F. Valeur and G. Vigna. “On the Detection of Anomalous System Call Arguments”. *ESORICS*, Oct. 2003.

[10] M. Bernaschi, E. Grabrielli, L. Mancini. "Operating System Enhancements to Prevent the Misuse of System Calls", In *Proceedings of the ACM Conference on Computer and Communications Security*, pp. 174 – 183, 2000.

[11] D. Kang, D. Fuller, and V. Honavar. “Learning classifiers for misuse and anomaly detection using a bag of system calls representation”. In *Proceedings of 6th IEEE Systems Man and Cybernetics Information Assurance Workshop (IAW)*, pp. 118-125, 2005.

[12] T. Bowen, M. Segal, and R. Sekar “On preventing intrusions by process behavior monitoring”. In *Proceedings of the Workshop on Intrusion Detection and Network Monitoring*, 1999.

[13] URL: http://www.metasploit.com

[14] Kurt Jensen. **Coloured Petri nets (2nd ed.): basic concepts, analysis methods and practical use**, volume 1, Springer-Verlag, Berlin, 1996

[15] A. Volynkin and V. Skormin. "Large-scale Reconfigurable Virtual Testbed for Information Security Experiments," in *Proceedings of the 3rd International Conference on Testbeds and Research Infrastructures for the Development of Networks and Communities*, Orlando, FL, May 21-23, 2007.

[16] S. Stolfo, W. Lee, E. Eskin. "Modeling system calls for ID with Dynamic Window Sizes", In *Proceedings of the DISCEX II*. June 2001.

[17] T. Bowen, M. Segal, and R. Sekar "On preventing intrusions by process behavior monitoring". In *Proceedings of the Workshop on Intrusion Detection and Network Monitoring*, 1999.

[18] D. Malan and M. Smith "Exploiting Temporal Consistency to Reduce False Positives in Host-Based, Collaborative Detection of Worms" In *Proceedings of the ACM Workshop on Recurring Malcode,* 2006

[19] A. Tokhtabayev and V. Skormin, "Non-Stationary Markov Models and Anomaly Propagation Analysis in IDS" *In proceedings IAS'07*, Manchester, England, August 2007.

[20] J. Gottman, R. Kumar. "Sequential analysis. A guide for behavioral researchers", *Cambridge: Cambridge University Press*, 1990.

[21] M. Kurant, A. Markopoulou and P. Thiran, "On the bias of BFS (Breadth First Search)", *International Teletraffic Congress (ITC 22)*, 2010

[22] Holger Ebel, Lutz-Ingo Mielsch, and Stefan Bornholdt "Scale-free topology of e-mail networks", *PHYSICAL REVIEW Edition 66*, 035103(R) 2002

[23] Jean-Pierre Eckmann, Elisha Moses, and Danilo Sergi "Entropy of dialogues creates coherent structures in e-mail traffic*" , PNAS,* 2004

[24] William Aiello, Fan Chung and Linyuan Lu "A Random Graph Model for Massive Graphs", *STOC*, 2000

[25] Watts, D. J. and S. H. Strogatz. Collective dynamics of 'small-world' networks. *Nature* 393440-42, 1990

[26] Joshua R. Tyler, Dennis M. Wilkinson, Bernardo A. Huberman "Email as Spectroscopy: Automated Discovery of Community Structure within Organizations", *Communities and Technologies*, 2003

7.0 LIST OF SYMBOLS, ABBREVIATIONS, AND ACRONYMS

AFRL	Air Force Research Laboratory
DCA	Dynamic Code Analyzer
IDS	Intrusion Detection System)
CCAPP	a non-graphical Norton Antivirus Process
LSASS	Local Security Authority Subsystem Service
AFOSR	Air Force Office of Scientific Research
API	application programming interface
HLL	high-level-language
PED	Propagation Engine Detector
CPN, CP-net	Colored Petri nets
FSM	Finite State Machine
ED&E	Executable Download and Execute